UNIVERSITY OF CALIFORNIA
Irvine

Hybrid Analysis Techniques for Software Fault Detection

Technical Report 89-26

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Michal Terry Young

Committee in charge:

Professor Richard N. Taylor, Chair
Professor Leon J. Osterweil
Professor Debra J. Richardson

1989

Contents

List of Figures

List of Tables

Acknowledgements

I am deeply indebted to Richard N. Taylor, my advisor. He always made himself available when I needed guidance, technical and otherwise. He supplied a salary, travel, equipment, and the assistance of two professional programmers to accelerate my progress. I am also grateful to my committee, Debra J. Richardson and Leon J. Osterweil, for reading the dissertation and offering valuable technical advice. Rami Razouk has also been an important source of technical guidance.

Kari Forester and Debra Brodbeck built most of the prototype system described in Appendix A and contributed to its design. I am fortunate indeed to have had such able assistance.

It has been a privilege to work and play with participants in the Arcadia software development environment project, at UCI and at the other Arcadia sites.

My companion and partner in all things is Cynthia Wenks. She kept me happy and sane during my years as a student.

During the period of this research I was supported as a research assistant by National Science Foundation grants CCR-8451421 and CCR-8521398 to Richard N. Taylor, and by grants from Hughes Aircraft (PYI program) and TRW (PYI program). Additional support came from NSF grant CCR-8704321 to the Arcadia project, with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6108, program code 7T10). A Coordinated Experimental Research grant from NSF transformed the hardware environment of the ICS Department and made much of this work possible.

I am grateful to the Association for Computing Machinery for permission to reprint material appearing in Chapters 1, 2, and Appendix A. Portions of those chapters originally appeared in "Rethinking the taxonomy of fault detection techniques," *Proceedings of the Eleventh International Conference on Software Engineering*, May 1989; "How to leave out details: Error preserving abstractions of state-space models," *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification,* July 1988, and "Integrated concurrency analysis in a software development environment," *Proceedings of ACM SIGSOFT '89: Third Symposium on Software Testing, Analysis, and Verification,* to appear in December 1989.

I am grateful to IEEE Computer Society for permission to reprint "Combining static concurrency analysis with symbolic execution," which originally appeared in *IEEE Transactions on Software Engineering,* October 1988. Portions of that paper appear in Chapters 3 and 4.

Curriculum Vitae

May 20, 1956	Born Eugene, Oregon
1983	B.A. in Computer Science, University of Oregon, Honors College, Summa Cum Laude.
1985	M.S. in Information and Computer Science, University of California, Irvine
	Dissertation: *Hybrid Analysis Techniques for Software Fault Detection*

Publications

Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

Michal Young, Richard N. Taylor, and Dennis B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.

Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated Concurrency Analysis in a Software Development Environment. To appear in *Proceedings of ACM SIGSOFT '89: Third Symposium on Software Testing, Analysis, and Verification,* Key West, December 1989.

Michal Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the Eleventh International Conference on Software Engineering*, Pittsburgh, May 1989.

Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, November 1988.

Michal Young. How to leave out details: Error-preserving abstractions of state-space models. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 63–70, Banff, Canada, July 1988.

Abstract of the Dissertation

Hybrid Analysis Techniques
for Software Fault Detection

by

Michal Terry Young

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1989

Professor Richard N. Taylor, Chair

Since the question "Does program P obey specification S" is undecidable in general, every practical software validation technique must compromise accuracy in some way. Testing techniques admit the possibility that a fault will go undetected, as the price for quitting after a finite number of test cases. Formal verification admits the possibility that a proof will not be found for a valid assertion, as the price for quitting after a finite amount of proof effort. No technique so dominates others that a wise validation strategy consists of applying that technique alone; rather, effective validation requires applying several techniques.

Michal Young, Richard N. Taylor, Dennis B. Troup, and Cheryl D. Kelly. Design principles behind Chiron: A UIMS for software environments. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 367–376, Singapore, April 1988.

Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. In *Proceedings of the First Workshop on Software Testing*, pages 170–178, Banff, Canada, July 1986.

Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wileden, and Michal Young. Arcadia: A software development environment research project. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, pages 137–149, Miami, Florida, April 1986.

Richard H. Granger, Jr., Jeffrey C. Schlimmer, and Michal Young. Components of experimental learning. In Joel L. Davis, Robert W. Newburgh, and Edward J. Wegman, editors, *Brain Structure, Learning, and Memory*, number 105 in AAAS Selected Symposia, pages 193–231. Westview Press, Boulder, CO, 1988.

Introduction

A variety of techniques for detecting faults in software have been developed [MH81, ABC82], and new techniques and refinements continue to appear [Hau84, TAV86, TAV88]. Fundamental undecidability properties of software make it unlikely that any single technique will be a panacea. Rather, each technique has particular strengths and weaknesses, and practical validation requires a combination of techniques. Growing support for the position that formal verification and program testing are complementary aspects of an integrated validation regimen is evidenced by the evolution of a workshop on program testing [TAV86] to a symposium on testing, analysis, and verification [TAV88]. A few proposals for cooperative, or hybrid techniques have appeared [Tay84a, Ost84]. This dissertation advances understanding of synergistic combinations of fault detection techniques, especially for concurrent software, by

- Introducing an improved classification scheme for considering the strengths and weaknesses of techniques and a framework for identifying ways of combining them,
- Providing the necessary theoretical underpinnings for establishing the soundness of certain combinations of techniques, and
- Describing a particular hybrid technique for analysis of concurrent software, in accordance with the framework and theory.

These contributions are described in more detail below.

Background

Software may contain *faults,* which may cause *errors* in executions of that software, possibly manifested in *failures* in the system of which the software is part. (The terms *fault* and *error* are sometimes used in these senses, and sometimes interchanged, in the testing literature.)

A fault is a discrepancy between a specification and its realization. It is meaningless to speak of faults (or errors, or failures) in the absence of a specification of

intended behavior, although some specifications may be implicit. For instance, since deadlock is almost always undesirable, it is convenient to consider absence of deadlock an implicit requirement of concurrent systems, even when not explicitly mentioned in a specification document.

Fault detection is embedded in a software development process, in the context of either product improvement or quality assurance. Fault detection for product improvement is aimed at finding and removing faults, while fault detection for quality assurance seeks to justify confidence of the absence or rarity of faults (or, in some cases, rarity of failures). This research focuses on finding faults so they can be removed for product improvement. In this endeavor, it is useful to make the simplifying assumption that all faults are equally important, and the goal is to find and eliminate as many as possible.

Techniques for justifying confidence (e.g., reliability growth models) are outside the scope of the research, as are techniques that differentiate between faults on the basis of severity (e.g., software safety). Non-statistical techniques for evaluating validation techniques (e.g., test coverage criteria) fit naturally in the framework.

Necessity of compromise. The impossibility of a perfect fault detection technique is simple to state and defend, but the import of this proposition is seldom fully grasped. Perfection in one aspect of fault detection can only be gained at the expense of imperfection in another. For instance, it is sometimes stated that static analysis techniques are "stronger" than dynamic analysis techniques, because the former can positively rule out certain classes of errors. How can this be? Of course, it must be that strength in ruling out errors is bought at the expense of a corresponding weakness in recognizing the absence of errors, i.e., a propensity to reject correct programs. As static analysis techniques are generally machine-aided proofs of particular program properties, it is not surprising to find that formal proof of correctness suffers from the same problem, and will continue to suffer from that limitation regardless of progress in proof techniques.

Even when the class of specifications is limited to decidable properties, or an analysis technique is allowed to err in a limited way, serious complexity problems remain. Mutation testing, for instance, suffers from combinatorial explosion in the number of mutant programs that must be tested.

Deadlock detection in concurrent programs is an example of an inherently complex problem in fault detection. Detection of potential deadlocks in concurrent CSP and Ada programs is exponential in the number of tasks, even with the simplifying assumption that the number of tasks is fixed. Since the problem is known to be NP-hard [Tay83a], the exponential complexity remains *regardless* of whether analysis is

performed using a flowgraph model of computation as described by Taylor [Tay83b], a Petri net model as prescribed by Mandrioli et al. [MZGT85], or some entirely different algorithm.

Most lower bounds on complexity, and in particular the lower bound results on complexity of checking deadlock, are established by an argument of the following form:

> To determine whether an object X (of a certain class), of size $|X|$, has property P is known to require at least $f(|X|)$ steps in the worst case.
> Any object X can be transformed into an object Y of size at most $g(|X|)$ in at most $g(|X|)$ steps. Moreover, object Y has property P' iff object X has property P.
> Let $h(|Y|)$ be an upper bound on the complexity of determining whether an object Y has property P'. If $h(g(|X|))$ were less than $f(|X|)$, then the previously known lower bound would be contradicted; therefore $h(g(|X|)) \geq f(|X|)$.

In the case of deadlock checking, X is drawn from a class of programs or models of programs, and Y is a decision problem known or believed to require exponential time (e.g., 3-CNF satisfiability). Since g is a polynomial in $|X|$ and f is an exponential, h must be an exponential. An analysis that is polynomial in the worst case must sidestep the above argument. It can do so in one of the following ways:

- Further restrict the class of programs that may be analyzed.
- Further restrict the class of specifications to be checked, i.e., check a simpler property.
- Announce the possibility of deadlock in programs where deadlock is impossible, when the exponential algorithms would correctly deduce the absence of deadlock.
- Announce the absence of deadlock in programs that may deadlock, when the exponential algorithms would correctly deduce the possibility of deadlock.

The first two approaches invalidate the lower-bound argument by contradicting its assumptions. Restricting the class of programs to be analyzed allows "hard" problems to be rejected (not all X can be transformed to Y in the restricted class). Restricting specifications is similar (property P' does not allow inferring property P).

Since the lower bound argument says nothing about how long it takes to get a wrong answer, the latter two approaches may also escape the lower bound. A degenerate technique could always announce a possibility of deadlock, regardless of the program to be inspected, in constant time. The first two approaches are subsumed by

the latter two: An analysis technique may always accept (or always reject) programs outside a restricted class, or it may substitute a decision about property Q for the desired decision about property P.

As a consequence of fundamental limitations on fault detection, it will always be the case that each available validation technique has its own weaknesses. It is unlikely that good software engineering practice will ever involve using a single "best" validation technique in all situations. A major difference between contemporary and medieval engineering of physical artifacts is that the modern engineer can depend on accurate assessments of the strengths and limitations of materials. The science of materials allows the engineer to choose steel-reinforced concrete for one building, wood for another, and moreover to have high confidence that the material chosen in each case is suitable. Software engineering has a very long way to go before validation techniques can be chosen with such confidence. Only the first tentative steps toward a systematic characterization of the strengths and limitations of validation techniques have been taken.

Contributions

This dissertation advances understanding of synergistic combination of fault detection techniques, especially for concurrent software. The primary contributions are:

- A framework within which the contribution of fault detection techniques can be discussed. The contribution of each technique is characterized by effort/accuracy tradeoffs inherent in that technique. This framework extends the conventional dichotomy of techniques as *static analysis* on the one hand, and *dynamic analysis* on the other. The conventional dichotomy is replaced by a more useful distinction between *sampling* of states (with resulting *optimistic inaccuracy*), and *folding* of states (with resulting *pessimistic inaccuracy*).
- Formalization of a relation of *error-preserving abstraction* between techniques that fold a state space. The error-preserving abstraction relation ensures that errors are not hidden by a simplification of a model, although spurious errors may be introduced. Abstraction functions may be obtained in several ways, including abstract interpretation (folding by summarizing data values) and analysis of modules in isolation (folding away information about the environment of a module). Sufficient conditions are given for proving that a particular abstraction is error-preserving with respect to a specification expressed in a (linear or branching time) propositional temporal logic. Properties of error-preserving abstractions provide a sound theoretical basis for combining pessimistic techniques, in accordance with the general framework.

- A hybrid fault detection technique that demonstrates the above-mentioned principles. The starting point is static concurrency analysis [Tay83b], a reachability analysis technique for detecting deadlocks and other anomalies in concurrent programs. Static concurrency analysis suffers from combinatorial explosion that currently limits its application to small programs, and on the other hand also suffers from spurious error reports (pessimistic inaccuracy). Both problems are addressed by appeal to the principle of *folding* a state-space in accordance with the principles of error-preserving abstraction. The combinatorial explosion problem is addressed by parceling large programs into modules; analysis of a module in isolation can be justified as a way of folding details of the system outside the module under analysis. The problem of spurious errors is addressed by combining static concurrency analysis with symbolic execution. The combined technique is sound because, for a large class of concurrency-related errors, static concurrency analysis is an error-preserving abstraction of symbolic execution and both are error-preserving abstractions of actual execution. While not yet incorporated in a practical tool, the concepts are demonstrated with a combination of some actual implementation, some detailed design, and some hand-worked examples.

Overview of the dissertation

A framework for combining techniques

Chapter 1 describes a classification scheme for fault detection techniques. Before one can discuss combining fault detection techniques in a useful way, one must have a general framework for comparing the contributions and weaknesses of each technique. The conventional taxonomy, characterized by a central dichotomy between *static* and *dynamic* analysis, is poorly suited to this end. We propose a modified classification scheme that highlights cost/accuracy tradeoffs inherent in every fault detection technique. This cost/accuracy tradeoff is the basis of principles for combining techniques.

State space analysis

A framework for evaluating the strengths and limitations of various validation techniques is a prerequisite for recognizing complementary techniques and combining them. A useful perspective (certainly not the only useful perspective) for evaluating techniques is to consider behavioral models of software as generators of state spaces. A *model* of an artifact is a representation that exhibits some interesting properties of

the original artifact, but which may differ in other properties. For instance, the "uses" hierarchy [Par79] is a model of a software artifact. By a *behavioral* model of software, we mean a model that emphasizes states and state changes during program execution. This is in contrast, for instance, to a denotational model in which states and state changes are de-emphasized. Narrowing the scope of our research to fault detection techniques based on behavioral models excludes some techniques from consideration, but a large family of techniques can be characterized in terms of the sets of behaviors they explore. Execution of software (acting as a model of itself) belongs to this family, along with analysis of various simplified models (Petri nets, flow graphs, so-called executable specifications, etc.)

A sequence of possible states, punctuated by state changes, is a behavior. The set of possible behaviors generated by a particular behavioral model is the *state space* of that model. A state space may be equivalently conceived as a set of behaviors, as a directed graph of reachable states, or as a (possibly infinite) tree of reachable states. State space analysis techniques compare a state space to a specification of allowable behaviors. The strengths and weaknesses of a technique depend on the class of models it can analyze, the class of specifications it can check models against, the computational cost of analysis, and the trustworthiness of the results.

Folding An analysis technique may abstract away some details of program execution, either because they are of no consequence or because removing them makes the model easier to analyze. Often, ignoring or removing details has the effect of *folding* states. That is, the technique uses a smaller state space than the actual program, and each state in the model state space represents several states in the normal program execution. In fact, if exhaustive enumeration of the state space is to be practical, a finite number of model states must represent an infinite number of program states.

In techniques based on program texts, or information derived from program texts such as flowgraphs, the degree of folding will generally be determined by the class of model. For instance, many techniques model control flow and omit data, thus folding together program states that differ only in variable values. Abstract interpretation [AH87] folds a large set of data values into a smaller set. Other foldings are possible, e.g., analyzing a module in the context of a simplified model of its environment.

In a folded model, some impossible behaviors (infeasible paths) as well as all possible behaviors of the original software artifact are represented. The impossible behaviors may trigger spurious error reports from an analysis technique. Reporting errors which cannot actually occur is called *pessimistic inaccuracy,* and is a general characteristic of techniques which employ folding.

Sampling A simple strategy for dealing with an infinite state space is to explore only part of it. Many techniques do just this. In particular, since the set of actual computation states of most programs is infinite, all techniques normally grouped under the rubric of *dynamic analysis* sample the state space of program execution.

Failure to report an error that may occur is called *optimistic inaccuracy.* The well-worn admonishment that program testing can reveal the presence of errors but not their absence derives from the fact that a sampling technique may admit optimistic inaccuracy. Pessimistic inaccuracy cannot occur in a technique that applies only sampling (and no folding) unless normal execution is disturbed in some way (e.g., if execution of a real-time program were slowed enough to miss a deadline). Much of the testing literature, therefore, is concerned with determining when a sufficiently representative portion of the state space has been explored to merit some confidence in the unexplored portions.

Combining fault detection techniques

Characterization of fault detection techniques in terms of folding and sampling, and consequent optimistic and pessimistic inaccuracy, can provide guidance for devising hybrid analysis techniques and integrated approaches to software fault detection. Folding techniques are subject to (at least) pessimistic inaccuracy, and sampling techniques are subject to (at least) optimistic inaccuracy. Some techniques employ both sampling and folding, and are subject in some degree to both sorts of inaccuracy. One may combine fault detection techniques to partially overcome these inaccuracies, in which case the primary concern must be the nature and extent of inaccuracy of each technique.

Several strategies for combining techniques are described in Chapter 1. The framework described in that chapter taxonomizes fault detection techniques by several attributes, and notes that many techniques in the literature leave some of these attributes unspecified. A first step, then, is to fully specify an analysis method by combining techniques to fully determine each attribute. Strategies for balancing effort and accuracy include using pessimistic techniques to concentrate optimistic techniques, using a more pessimistic technique to generate candidate errors for examination by a less pessimistic (more accurate) technique, and combining coverage criteria (e.g., functional and structural testing) for optimistic techniques.

Theoretical basis

Chapter 2 provides the necessary theoretical underpinnings for considering combinations of pessimistic analysis techniques. The proposed framework for combining fault detection techniques is based on a distinction between optimistic inaccuracy and pessimistic inaccuracy. Many potential combinations depend on showing that a computationally cheaper technique detects every fault that would be detected by a more expensive technique, i.e., that any additional inaccuracy of the cheaper technique is only in the pessimistic direction. While pessimistic inaccuracy results generally from folding a state space, this rule of thumb is not precise enough to guide application of the framework. A formal statement of the conditions under which folding introduces only pessimistic inaccuracy is required.

Chapter 2 defines a relation between state space models called *error-preserving abstraction.* This relation makes precise the notion that a particular simplification folds a state space in a manner that introduces inaccuracy only in the pessimistic direction. Error-preserving abstraction is necessarily relative to the class of errors an analysis technique is designed to detect. The focus is on sequencing constraints appropriate for reactive systems, and particularly for concurrent programs. Sufficient conditions for establishing the error-preserving property are related to formulas of propositional temporal logic. These conditions are stated in the form of lemmas (the overall theorem to be proved being the error-preserving abstraction relation between particular analysis techniques).

The set of possible behaviors of a program is conceived of as a directed graph in which nodes represent program states and edges represent state transitions (events). A state-space *model* of program execution is similarly conceived of as a directed graph.[1] A simplification (folding) of a model is naturally conceived of as a relation between graphs. The nature of this relation determines whether errors in one graph are represented in the other.

An abstraction is formalized as a function that maps nodes (states) in one directed graph to nodes in the other, simplified graph. It is formalized as a (total) function, rather than a general relation, to ensure that every state in the domain is represented in the range. However, it is not required to be one-to-one or onto. This distinguishes our approach from the similar notion of *projection* [LS84]. A projection is a mapping from one state-space to another in which, like a geometric projection

[1]To be really precise, we ought to distinguish between model of a program and a model of program behavior. A behavioral model of a program (e.g., a flowgraph or Petri net) generates a state space that is a model of program behavior. The state space may be implicit, or it may be explicitly represented (as, e.g., in the reachability graph of a Petri net). The relation discussed here is between models of behavior, but one normally reasons indirectly about the relation between models of behavior by considering models of the program.

from a graph embedded in n-space to a hyperplane of $n - k$ dimensions, every node (and edge) in the range corresponds to one or more nodes (edges) in the domain. A characteristic of practical simplifications, on the other hand, is that they often introduce *unexecutable paths,* i.e., nodes and edges that do not occur in the original state space.

Sufficient conditions for proving that an abstraction is error-preserving are developed in two stages, for pedagogical reasons. First, a set of conditions relative to global safety properties expressed in propositional logic is described. More interesting is an additional set of conditions that are imposed when one expands the class of specification formulas from standard propositional logic to propositional temporal logic. Temporal logic is suited for specifying the behavior of reactive systems, and is particularly useful in concurrent systems where even the propositional form of the logic can be used to specify non-trivial properties. Furthermore, the branching-time version of propositional temporal logic can be efficiently checked against directed graph representation of program behavior [CES86], and this capability has been incorporated in practical analysis tools for concurrent software [FRV85, MR87].

In Chapter 2, conditions for establishing an error-preserving abstraction are related to the CTL* logic of Emerson and Halpern [EH83]. This logic combines linear and branching time in a single system. Most of CTL* is handled, including all of its linear-time subset except for the "next-time" operator. The "next-time" operator allows construction of statements like "a will be true in the forty-seventh state," violations of which are not amenable to preservation by non-trivial abstractions.

A hybrid analysis technique

Static concurrency analysis is an analysis technique for detecting anomalous synchronization patterns (deadlocks, misuse of shared variables, etc.) in concurrent programs. In addition to errors that may actually occur, it may also report spurious errors involving infeasible execution paths. Moreover, combinatorial explosion limits application of the technique to small programs and programs that obey severe restrictions in synchronization structure. Both of these problems can be ameliorated through application of the theory of error-preserving abstractions within the framework of hybrid analysis techniques.

Chapter 3 describes ways to overcome complexity problems in static concurrency analysis by constructing parceled models which are error-preserving abstractions of the complete program. The class of systems that can be effectively parceled for analysis (limiting each parcel to a size that can be practically analyzed) is expanded by allowing each partition to be an error-preserving abstraction of the whole, rather than a projection as in earlier approaches. Chapter 4 shows how pessimistic inaccuracy in

static concurrency analysis can be ameliorated in a hybrid technique. Spurious error reports are reduced by integrating concurrency analysis with symbolic execution, sharpening the results of the former without incurring the full costs of the latter applied in isolation. The soundness of this combination depends on the fact that static concurrency analysis is an error-preserving abstraction of symbolic execution, with respect to the class of faults it is designed to detect.

Appendix A describes a design and prototype implementation of a tool to support static concurrency analysis. Modular design to support integration with symbolic execution and other analysis capabilities in a software development environment is emphasized.

Chapter 1
Framework

1.1 Introduction

The goal of this dissertation is to advance understanding of how different fault detection techniques (analysis, testing, and verification) can be combined. To consider how techniques are to be combined, one must have a framework — a taxonomy, or classification scheme — for characterizing strengths and weaknesses and opportunities for combination. In this chapter we examine a conventional taxonomy and find it wanting with respect to our goals. A revised classification scheme is proposed.

Software validation techniques are usually classified as *dynamic analysis* if they involve program execution, or *static analysis* otherwise. This dichotomy serves a useful purpose in planning validation activities, if fault detection techniques are considered in isolation. But a thorough validation regimen incorporates several fault detection techniques, and it is important to consider their interactions. The static/dynamic distinction is not very helpful in this regard.

Every practical fault detection technique necessarily embodies a delicate balance of accuracy and effort. The design tradeoffs made in achieving this balance are more useful in elucidating relations between techniques and taking advantage of their interactions than the static/dynamic distinction. A particularly useful distinction is between state-space analysis techniques that *fold* actual execution states together (to make the state space smaller or more regular) and those that explore only a *sample* of possible program behaviors. The strategies of folding and sampling result in different sorts of inaccuracy (*pessimistic* and *optimistic,* respectively), which are sometimes erroneously equated with the static/dynamic distinction.

1.2 The conventional taxonomy

Software modeling and analysis techniques are commonly taxonomized according to their operational characteristics. A central dichotomy in these taxonomies is the distinction between *static analysis,* which does not require program execution, and *dynamic analysis,* which does. For example, the taxonomy of techniques presented in a popular IEEE tutorial [MH81] is organized essentially along two dimensions, one being static versus dynamic analysis and the other being the type of documents (requirements, design, or source code) used by the technique (see Table 1.1).

This conventional taxonomy is well suited to the practical end of planning a series of validation activities in a project. It identifies the sort of documents required for each technique, and thereby allows the project manager to determine where the technique may fit in a project's life cycle. The distinction between static and dynamic analysis also serves this planning purpose, since execution generally requires a piece of completed code. (The whole system is not necessarily required, but if it is not available then some test scaffolding is needed in addition to the modules to be tested). Referring again to Table 1.1, the reader can see that the top two rows of the left column (static analysis of requirements and design documents) are activities that can be carried out before implementation. The top two rows of the right column (dynamic analysis using requirements and design documents) are activities for which preparation can be made in earlier life cycle stages, but which can only be completed in the implementation stage.

Sometimes the conventional taxonomy is also taken to indicate the relative cost of techniques, though this is misleading. A common rule of thumb is that static analysis is (computationally) cheaper than dynamic analysis. In fact, either dynamic or static analysis may be expensive or cheap, depending on the thoroughness and accuracy of the particular analysis technique. The characteristics in the proposed extension to the conventional taxonomy, described below, are a more reliable guide to computational cost.

The most significant inadequacy of operational characterization is seen, however, when attempting to formulate an integrated approach to software validation. Recognizing that no single technique is capable of addressing all fault-detection concerns, various attempts to define a comprehensive software validation scheme have been put forth, such as [Ost84] and [Tay84a]. The limited success of these attempts is due in part to their static analysis/dynamic analysis orientation. In particular, the operational taxonomy predisposes one to view each technique as applied in isolation (or in sequence) and obscures the more substantive concerns of technique interaction. In particular, every modeling or analysis technique involves some compromise between accuracy and completeness on the one hand, and tractability on the other.

	Static	*Dynamic*
Requirements	Informal checklists Formal modeling	Functional testing Testing by classes of input data Testing by classes of output data
Design	Static analysis of design documents	Design-based testing
Programs	General information Static error analysis Symbolic execution	Structural testing Expression testing Data-flow testing

Table 1.1: A conventional taxonomy of software modeling and analysis techniques, from [How81b] and [How81a].

The dimensions of this tradeoff are largely orthogonal to the issue of whether or not program execution is involved. These tradeoffs are explored in Section 1.3.

The shortcomings of operational classification are highlighted by the family of techniques known as symbolic execution, symbolic evaluation, or symbolic testing. These techniques do not fit clearly in either the static analysis or dynamic analysis category. Howden [How81b, How77] places symbolic testing among static analysis techniques, although conventional program testing is a special case of symbolic testing. In fact, variations on symbolic execution span the gamut from formal verification to testing. Section 1.4 describes in more detail the problem of lumping these techniques in the "static analysis" category, and how these problems are avoided by a revised categorization.

1.3 Analysis tradeoffs

Since the question "Does program P obey specification S" is undecidable for arbitrary programs and specifications, every fault detection technique embodies some compromise between accuracy and computational cost. It is important to grasp that the necessity of admitting inaccuracy does *not* arise out of limitations in the current state of the art. Rather, since the presence of faults is generally an undecidable property, it is not even theoretically possible to devise a completely accurate technique

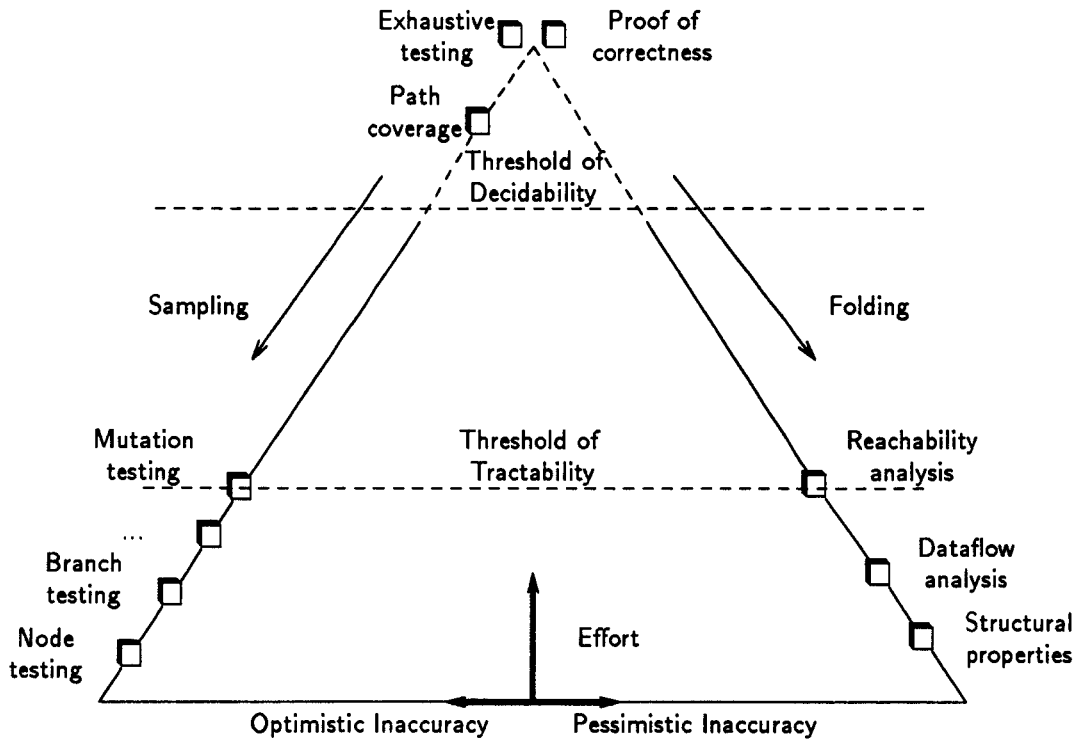


Figure 1.1: An intuitive view of tradeoffs between accuracy and effort.

that is applicable to arbitrary programs. Every practical technique *must* sacrifice accuracy in some way.

The conventional taxonomy of fault detection techniques captures some important dimensions of the analysis technique design space (particularly the relation between sources of information and the classes of faults detected by a technique). However, it does not adequately address tradeoffs between accuracy and computational effort.

Dimensions. The primary means available to reduce the complexity of analysis are *folding* states together (i.e., by abstracting away details) or *sampling* a subset of the state space. Figure 1.1 summarizes (and considerably oversimplifies) the relationship between effort, folding, sampling, and inaccuracies. These relationships are the subject of the remainder of this section, and are the basis of our revision of the conventional taxonomy of fault detection techniques.

Since the purpose of a taxonomy is practical rather than theoretical, we treat computational effort as a continuum with no sharp division between the truly impossible and the merely hopeless. The "threshold of tractability" in Figure 1.1, which separates pragmatic techniques from theoretically possible but vastly expensive decision procedures, is more important than the imaginary "threshold of decidability" beyond which effective algorithms do not exist. (A more accurate representation of the design space would depict the "effort" axis stretching off to infinity, with guaranteed proof of correctness and exhaustive testing at infinite distance from the origin. Figure 1.1 compromises accuracy in order to fit the diagram on a finite page.)

In order to limit computational expense, a technique must admit at least one of two possible kinds of inaccuracy, *pessimistic* inaccuracy or *optimistic* inaccuracy. Pessimistic inaccuracy is failure to accept a correct program. Optimistic inaccuracy is failure to reject an incorrect program. Techniques that admit of pessimistic inaccuracy but no optimistic inaccuracy are sometimes called *conservative*. Optimism and pessimism are shown as opposite directions on a single axis in Figure 1.1 because most often optimism results from sampling and pessimism results from folding. In principle, and sometimes in practice, a single technique may suffer both sorts of inaccuracy.

Automatic construction of program proofs for arbitrary programs lies at the apex of the design space triangle, representing an ideal case of complete accuracy. Being infallible, it is allowed neither to construct an invalid proof, nor to fail to find a valid proof for a correct program. Such an infallible technique is, of course, impossible (because of the halting problem). The more reasonable hope of automatically constructing program proofs for *some* programs, or assisting a programmer in constructing proofs, is a pessimistic technique, because failure to construct a proof does not imply the program is incorrect.

Exhaustive testing shares the apex of the design space with infallible proof construction. Exhaustive testing is, in fact, a "proof by cases" of program correctness. Testing all executable paths through a program (which is already only a sample of the space of behaviors generated by all possible input data) is generally impossible because programs with loops have an infinite number of possible paths.

The reader may find it unintuitive to equate infinite effort for exhaustive testing with undecidability in program proving. To see that these are in fact the same problem seen from different perspectives, consider the following procedure for verification: Generate all theorems derivable, by valid proof rules, from axioms describing the program. This procedure is easily mechanizable. Syntactic variations can be generated in the same manner, by including appropriate rewrite rules among the valid proof rules. If any of the generated theorems is identical to the program specification, announce that the program has been verified and halt. Clearly, this procedure would

be sound *if* one could wait forever (literally) for an answer. Exhaustive testing is just the same.

Models of Execution. We have ignored (and will continue to ignore, for the most part) another difference between proving and testing. Notions of correctness, reliability, etc., are only meaningful with reference to a model of computation. Two rather different models are used in testing and proving, and they result in quite different notions of what validation means.

A theoretical, ideal machine perfectly obeys the semantics of a particular language in executing a program. This idealized machine may be quite different from any physical machine. The theoretical machine never has roundoff error or arithmetic overflow, or runs out of storage, or incorrectly interprets a program. Correctness with respect to such a model does not imply that a program will execute correctly on all, or even some, physical machines. In principle, each level of virtual machine (compilers, operating system, etc.) may be verified, as may the design of the hardware, although this is a mind-boggling task. Even if verification were completed at every level, correct execution rests ultimately on the behavior of physical objects that may be flawed or eventually wear out.

The alternative is to refer to a particular target machine on which a program is to run. This has the advantage that all levels of implementation may be validated at once. On the other hand, particular machines can have peculiarities that allow a program to run "correctly" on that machine, although it will fail on most other machines (and certainly on the ideal machine, which makes no promises except to obey the semantics of a language).

It is convenient to assume that the ideal machine is an accurate model of a particular physical machine (or vice versa) when comparing or combining fault detection techniques. Moreover, tradeoffs involving effort and accuracy are largely orthogonal to the particular model of computation used. Accordingly it will not be considered further in this paper.

1.3.1 Folding

An analysis technique may abstract away some details of program execution, either because they are of no consequence or because removing them makes the model easier to analyze. Often, ignoring or removing details has the effect of *folding* states. That is, the technique uses a smaller state space than the state space of the actual program, and each state in the model state space represents several states in the normal program execution. In fact, if exhaustive enumeration of the state space is

to be practical, a finite number of model states must represent an infinite number of program states.

Usually one wishes to guarantee that simplifications employed in analysis introduce only pessimistic inaccuracy, i.e., that no errors will be hidden. Such a guarantee can only be made relative to a class of specifications. For safety properties of individual states, it is sufficient to show that each potential "bad" state in normal[1] program execution is represented by a "bad" state in the folded model. To preserve violations of specifications regarding paths in the execution state space, including liveness properties and precedence properties, additional conditions must be imposed on the mapping. A set of sufficient conditions for showing that a folding preserves violations of specifications expressed in propositional temporal logic are given in Chapter 2.

The degree of folding is generally determined by the class of model, and how it is derived from program text. For instance, many techniques model control flow and omit data, thus folding together program states that differ only in variable values.

It is also possible to fold instances within the same class of model (e.g., folding a Petri net model to create a simpler Petri net). For instance, in analyzing concurrent systems, a principle sometimes called "virtual coarsening" allows many sequential steps of a process to be combined when those steps are independent of other processes. Applications of virtual coarsening include reductions of control graphs by analysis tools in the SARA design environment [EFRV86], and reductions of program flowgraphs in the anomaly detection techniques of Taylor [Tay83b] and Long and Clarke [LC89].

Folding in program verification. In formal program verification one usually avoids explicitly constructing representations of program states. Instead, theorems describing properties of program behavior are derived. Since the set of theorems derivable from a program text taken together with a set of axioms and rules of inference expressing the semantics of a programming notation[2] is infinite, exhaustive enumeration of theorems is no more practical than exhaustive enumeration of execution states. The hierarchical structure of abstractions necessary for practical verification

[1] *Normal* execution may be execution on an ideal, theoretical machine, or execution on a particular machine, depending on the goals of analysis. When there is a mismatch between the goal and the "normal" model, both pessimistic and optimistic inaccuracies can result. For instance, if one is interested only in execution on a particular machine, a deadlock detection technique based on the semantics of a programming language may both report errors that cannot occur because of peculiarities of the process scheduler on the target machine (pessimism) and fail to uncover errors in the implementation of the language (optimism).

[2] This discussion is oriented toward axiomatic semantics and verification in the Floyd-Hoare style. Details of the relation between implicit and explicit representations of an execution state-space would differ if one chose, say, a denotational framework, but a similar relation could be drawn.

is exactly mirrored by a set of foldings of the execution state space, even when no explicit representation of the execution state space is constructed.

Consider a proof of a program involving an abstract data type (ADT). It is impractical to consider the implementation details of the ADT while verifying properties of the program as a whole. Instead, one first verifies properties of the ADT, and then verifies properties of the program as a whole in terms of the abstraction. A typical program verification will involve many steps of this type; a verification performed completely at the level of implementation details would be incomprehensible.

A representation invariant for an abstract data type folds the state space of the implementation into the state space of the data abstraction. Proof of the representation invariant is required to justify reasoning in terms of the folded state space. The creative part of verification is typically choosing the right abstractions, i.e., determining how to fold the space to make the next inference step easy.

When an abstraction step is generally useful and does not require creativity to apply, it may be embodied in a completely mechanical procedure. Whereas formal verification involves creative processes, including many repeated foldings of the execution state-space, automatic techniques such as data flow analysis or reachability analysis apply a single folding. The soundness of this folding must be established once and for all in justifying the analysis technique. A simple example is static type checking. Using formal verification techniques, it would be possible but challenging to prove that a particular Lisp program, with no type declarations, never attempted to take the tail of an integer nor divided a list by a floating point number. By requiring that each variable take on values of only a single, declared type, strongly typed languages make the verification simpler — so simple that it can be completely mechanized and routinely carried out by compilers.

1.3.2 Sampling

A simple strategy for dealing with an infinite state space is to explore only part of it. Many techniques do just this. In particular, since the set of actual computation states of most programs is infinite,[3] all techniques normally grouped under the rubric of *dynamic analysis* sample the state space of program execution.

[3]It is actually possible to fix a finite bound on the state space of program executions on any particular hardware. Since the number b of bits of storage available to a program, must be finite (infinite tapes exist only in theoretical machines), the program can be in only 2^b distinct states. Even for small machines this number is greater than the number of nanoseconds since the big bang, so it is practical to consider it infinite.

The well-worn admonishment that program testing can reveal the presence of errors but not their absence derives from the fact that a sampling technique almost always admits optimistic inaccuracy.[4] Pessimistic inaccuracy in a technique based on sampling the actual execution state space is impossible unless normal execution is disturbed in some way (e.g., if execution of a real-time program were slowed enough to miss a deadline). Much of the testing literature, therefore, is concerned with determining when a sufficiently representative portion of the state space has been explored to merit some confidence in the unexplored portions. (Contrary to what Figure 1.1 may suggest, common coverage metrics are only partially ordered with respect to optimistic inaccuracy; see, for instance, [CPRZ85] and [FW86].)

Sampling is not limited to techniques that explore the state space of normal program execution (conventional testing). All varieties of symbolic execution, for instance, fold states together by representing a large number of actual data states by a smaller number of symbolic data states. Many varieties of symbolic execution explore only a portion of the resulting state space, because although "smaller" it is generally still infinite. (Varieties of symbolic execution are considered in more detail in Section 1.4.) Some models with finite but large state spaces also rely on sampling. For instance, a Petri net may be exhaustively analyzed in some cases, but when exhaustive analysis is impossible, Petri net simulation may be used [Raz87].

Choosing samples. In classical testing of programs, program behavior is controlled by input data. Thus, a sample of program behaviors is chosen indirectly through a choice of test data. Uniform sampling of the input space (random testing) generally projects onto a very non-uniform sample of the space of possible execution states; consider:

```
if i = j then
  Do something wrong
else
  Do the right thing
end if;
```

If i and j are integers, and are inputs to the program, then random selection of inputs has an infinitessimal chance of exercising the erroneous behavior.

Test adequacy criteria are designed to ensure that the behaviors chosen are appropriately distributed to increase the likelihood of revealing errors. Often this is accomplished by partitioning behaviors into classes, and requiring samples to be

[4] Cases in which sampling may be relied upon without optimistic inaccuracy are limited to very restricted classes of programs. For instance, Howden [How85] has shown that a finite set of test points suffices to establish equivalence of polynomials.

drawn from each class. Specification-based selection criteria apply directly to test data, but explore different parts of the state space (or reveal missing program logic) by testing classes of data that must be treated differently or are often treated differently (e.g., boundary values and special values [How80]). Control flow and data flow coverage criteria [RW82, CPRZ85, FW86] relate more directly to the program execution state space. These structural criteria for sampling can be related to a folded model of execution. A control flow coverage criterion (node coverage, branch coverage, etc.) can be related to a flowgraph model in which data values are omitted and only control points distinguish states; data flow coverage criteria can be related to an extended flowgraph model in which limited aspects of the data state are retained. The coverage criterion is satisfied if each state in the folded model is represented by at least one state in the sample.

Hamlet pointed out in [Ham87] that an appropriate distribution depends on the purpose of testing. For reliability estimates, an operational distribution of inputs is required (but often impossible to obtain). For confidence in "probable correctness," appropriate samples must be drawn from the space in which faults are distributed, which is more closely related to the textual space of the program. The results of probable correctness theory are so far mostly negative; [HT88] shows that partitioning (which includes all functional and structural coverage criteria) is not an improvement over random testing for achieving high statistical confidence in program correctness. On the other hand, partitioning may be as close to uniform sampling as one can achieve in practice.

An alternative to both reliability estimates and probable correctness is confidence in the absence of particular faults. Several fault-based and error-based criteria for selecting test data have been put forward. Mutation testing is most direct: A sample of program behaviors is judged adequate when it differentiates between the actual program and a set of programs with hypothetical faults [Bud81, DGM+88]. Other fault-based methods avoid explicitly creating alternative programs, but attempt to select test data that would reveal a particular class of faults if it were present. In the fault-based approaches of Morell [Mor88] and Richardson and Thompson [RT88], this determination is made with the aid of symbolic evaluation, a folded model.

1.4 Example: Symbolic evaluation

The clearest example of the inadequacy of the conventional static/dynamic dichotomy is the group of techniques known collectively as symbolic evaluation. These techniques are conventionally classified as static analyses, because they do not involve normal program execution. Two symbolic evaluation techniques that share a

representation of a program state, and very little else, are described below. The differences between these techniques, their capabilities, and their shortcomings illustrate the problems inherent in lumping them together in a taxonomy of fault detection techniques. The revised taxonomy reveals that, while both techniques employ some folding, one folds the state space further to allow exhaustive enumeration of program behaviors, and the other visits only a sample of the complete space of possible states.

The two techniques described here are *symbolic execution* and *global symbolic evaluation.* The description of symbolic evaluation methods here differs in detail from descriptions in the literature, in order to simplify the presentation and highlight the state-space perspective. We limit attention to programs containing only assignment statements, *while* loops, and *if* statements, and ignore procedure calls and input/output. The interested reader can find thorough introductions to theoretical and practical aspects of symbolic evaluation in [HK76] and [CR81], respectively.

1.4.1 Symbolic execution

The model schema used by symbolic execution is a program flowgraph, with nodes for each executable program statement. An additional node is placed before the first executable statement, and one after each terminal node. *If* statements and *while* loops are represented by nodes with two out-edges. A token is used to represent a thread of control. (For the current discussion, we assume a single thread of control.) Two additional pieces of information are maintained. The *path expression* associates program variables with symbolic values (algebraic expressions). The *path condition* is a predicate that describes the conditions necessary to follow a particular execution path.

Symbolic execution begins with a token on the edge leading into the first executable statement of the program. (We added a node before this statement so that execution could begin with a token on this edge.) The path condition is initially set to *true,* and the path expression associates each program variable with a unique symbol.

Analysis proceeds by advancing the token through a statement, and onto an edge leaving the statement. When a token is advanced through an assignment statement, the path expression is modified. For instance, if the assignment were $C := A + B$, then the current expression associated with C would be replaced by $\alpha + \beta$, where α and β are the current symbolic values of A and B, respectively.

Advancement of a token through a conditional branch node (*if* or *while*) adds a term to the path condition, corresponding to the branch chosen. For instance, if the branch condition were $A = B$, and the *true* branch were chosen, then $\alpha = \beta$

would be conjoined with the path condition, where α is the expression associated with A in the path expression, and β is the expression associated with B. If the *false* branch were chosen, then the complement of that predicate would be conjoined with the path condition. In either case, if the new path condition is inconsistent (provably equivalent to *false*), then the path is unexecutable.

States in symbolic execution are characterized completely by *(path expression, path condition)* pairs. (If non-deterministic choice were possible, the current token location would also be required.) For a program without loops, this state space will be a tree that could, in principle, be exhaustively explored by a reachability analysis technique.

For programs with loops, the state space will generally be infinite. Thus, exhaustive generation of states is ruled out. Instead, sampling can be used to explore only a portion of the state space. Symbolic execution starting from the initial state and progressing along some path to a terminal state is called *symbolic testing.*

Symbolic testing is like conventional program testing in most ways. It admits optimistic inaccuracy, since faults may lie on paths that are not explored. Symbolic testing also requires an oracle ("Is the final path expression acceptable?"), and like conventional program testing it may be practiced to some coverage criterion. The extent of optimistic inaccuracy is reduced because there is no sampling of possible data states encountered on a particular path, but pessimistic inaccuracy may be introduced if it is impossible to determine the outcome of an oracle using the symbolic representation of a path computation.

A practical difference between symbolic testing and conventional testing is that symbolic testing may be explicitly directed to follow a path, whereas normal program execution follows a path determined by the test data. In fact, symbolic execution systems have been used to derive test data that force normal execution to follow particular paths [Cla76].

1.4.2 Global symbolic evaluation

Whereas symbolic execution explores a sample of the state space, global symbolic evaluation folds the infinite state space of symbolic evaluation into a finite set of representative states. Thus, while symbolic execution is like conventional dynamic analysis techniques, global symbolic evaluation is like other static analysis techniques.

Global symbolic evaluation attempts to fold together states reached along paths that differ only in the number of iterations a loop is traversed. One approach to global evaluation is to represent the effect of a loop by a set of recurrences, which in some

cases can be simplified to closed form expressions [CHT79]. An alternative, especially suitable when symbolic evaluation is used as an aid to formal verification, is to cut each loop with an assertion. The latter approach is discussed here.

Imagine that every loop in a program is cut by a loop invariant assertion that lies on a flowgraph edge. Also, an edge leading into the first program statement is labeled with an assertion describing the domain of applicability of the program, and the edge leading out of each terminal statement is labeled with an assertion stating the required output condition of the program. An assertion is satisfied if, for every state in the state space of symbolic evaluation such that the token lies on an edge labeled by the assertion, the assertion can be proven from the path condition and path expression. If the path expression and path condition are not sufficient to prove the assertion, then we say the assertion is violated.

If every loop is cut by assertions, then every path through the flowgraph is made up of a sequence of subpaths between assertions. Every such subpath is finite, and there are a finite number of them. Analysis proceeds as for symbolic testing, but with one important difference: instead of following a path from the beginning of execution, each subpath from one assertion to the next is separately analyzed. For each such subpath, the path expression and path condition are initialized to reflect the initial assertion. At the final state along each subpath, the final assertion is checked against the path condition and path expression. If the assertion cannot be proven, an error is reported. If each individual subpath is accepted, then every path through the program must satisfy every predicate.

Note that what the loop-cutting assertions have done is to fold infinite sets of states into representative states by discarding details of the execution state. Each time an assertion is reached at the end of a subpath, the path expression and path condition may be different. Details of these different conditions are discarded, and only the information in the assertion is preserved. One may think of the assertions as filters that prevent too much information from passing through. (Used in this way, they also preclude derivation of a symbolic representation of a complete path, which is an advantage of methods that solve recurrence equations to derive closed forms.)

Unlike symbolic testing, global symbolic evaluation is a *pessimistic* technique. It does not accept incorrect programs (assuming, of course, that the input and output assertions fully and correctly capture program specifications). Pessimistic inaccuracy stems from the difficulty of finding satisfactory loop-cutting assertions. An unsatisfactory assertion will either be too strong (not provable at terminal states along partial paths) or too weak (not sufficient as an assumption to prove the next assertion along a subpath), or perhaps both. In fact, since successful global symbolic evaluation using the loop-cutting method is a machine-aided proof of partial correctness using the loop invariant method [HK76, KE85], finding satisfactory assertions is an unsolvable

problem in the general case (as is solving the recurrence equations encountered by global symbolic evaluation methods employing loop analysis).

1.4.3 Summary

Both symbolic testing (symbolic execution of particular program paths) and global symbolic evaluation employ some folding, namely representing whole classes of data by symbolic values. This degree of folding still leaves an infinite state space that cannot be exhaustively explored. Symbolic testing examines only a finite sample of this space, and thus incurs optimistic inaccuracy. Global symbolic evaluation folds the state space further, to obtain a finite number of representative states, and thus incurs pessimistic inaccuracy.

A taxonomy intended to facilitate combining analysis techniques in an integrated validation regimen must highlight the relative strengths and weaknesses of each technique. In the case of symbolic evaluation techniques, it should point up the optimistic inaccuracy of symbolic testing (and its similarity to conventional testing in that regard) and the pessimistic inaccuracy of global symbolic evaluation. A revised taxonomy that does just that is introduced in the following section.

1.5 A revised taxonomy

1.5.1 The taxonomy

The following taxonomy highlights the fundamental characteristics of state-space analysis techniques from the viewpoint of considering how they may be combined. The primary characteristic, not surprisingly, is the distinction between

- *state folding* and
- *state sampling.*

Additionally, the characteristics of

- *model schemata*,
- *representation of the state space*, and
- *oracle*

are called out, because of their practical utility in considering technique combination.

Having discussed folding and sampling at length, we briefly consider the three auxiliary characteristics.

Model schemata. A class of model schemata (Petri nets, flowgraphs, program texts in some language) determines a class of state spaces within which sampling or folding may occur. It is easiest to exploit interactions between techniques when the same model schemata is shared between them. For instance, Chapter 4 of this dissertation describes a method for combining static concurrency analysis with symbolic execution, based on the observation that they share an underlying flowgraph model of execution, and that the state space of the former is (conceptually) obtained from the latter by folding together states with different path expressions.

Representation of the state space. Some techniques explicitly represent the state space in the form of a *reachability graph,* while other techniques leave the state space implicit and represent only a single "current" state at any one time. Some techniques (notably test coverage metrics) keep a partial record of the portions of the state space visited. Some techniques, e.g. constrained expression analysis [ADWR86], infer properties of a state space without constructing any explicit representation of it.

Oracles. Among techniques that build explicit representations of a state space (whether partial or complete), one can often distinguish a component of the technique for exploring the state space from a procedure for determining whether a state or path is faulty. In the testing literature, for instance, coverage criteria are usually treated separately from test oracles. The notion of oracle is present in practically every technique, however, and so is called out here.

Important characteristics of oracles include:

- Is an oracle function explicit in the technique? Some techniques (notably test coverage metrics) assume the availability of an oracle, but do not describe how to build it.
- If the oracle is explicit, does it check individual states or paths? If it checks states, does it check only a subset of states (e.g., terminal states)?
- Does the oracle check for certain fixed properties (e.g., absence of deadlock) or may a class of properties be specified by the user? (When fixed properties are checked, these are usually *implicit specifications,* whereas explicit specification are supplied by the user.)

1.5.2 Sample derivative categories

These characteristics can be used to divide state space analysis techniques for fault detection into groups exhibiting similar properties in the critical dimensions.

Some, which seem to us to arise naturally, are flow analysis, reachability analysis, and testing. They are considered in turn below, in the light of the taxonomy's characteristics. The purpose of this section is not to give a rigorous grouping; it is just to illustrate how some familiar techniques appear in the light of the revised taxonomy.

Flow analysis.

Folding: Control flow is modeled, data values are ignored. Sampling: None.

Model schema: The model schema is a directed graph that can itself be considered as a state space, and the analysis procedure labels nodes in this graph with predicates. We include in this class both techniques derived from classical data flow analysis, e.g. [OO86], and other techniques that similarly label a directed graph, such as the temporal logic checking algorithms of Clarke, Emerson, and Sistla [CES86] and Fernandez, Richier, and Voiron [FRV85]. These techniques are procedures for *checking* a graph rather than for building it, so they may naturally be combined with other techniques for constructing a representation of the state space.

Representation of the state space: Explicit, given as input (the control graph).

Oracle: Hard-wired classes of anomalous behavior in the case of systems like DAVE [OF76], programmable and path-oriented in [OO86, CES86, FRV85].

Reachability analysis.

Folding: Control flow is modeled, data values are ignored.

Sampling: None.

Model schema: Various graph models, including Petri nets and program flowgraphs.

Representation of the state space: Finite, explicitly represented, and exhaustively enumerated (generated). Examples of reachability analysis include Petri net reachability analysis [Pet81], static concurrency analysis [Tay83b], and analysis in the "control" domain of UCLA graph models [EFRV86]. Because reachability analysis is exhaustive, all reachability analysis techniques *fold* the space of program behaviors into a finite state space, and thus are *pessimistic*. These are primarily techniques for constructing a representation of a state space.

Oracle: Although a procedure for checking particular properties may be hardwired into a reachability analysis technique, it is usually superior to employ a logically separate, "programmable" checking procedure. An example of this approach is the P-Nut system of Razouk and Morgan [Raz87, MR87], which combines Petri net reachability analysis with a checking procedure for branching time temporal logic assertions.

Conventional Testing.

Folding: Conventional program testing explores the state space of actual program execution with no folding.

Sampling: The state space is infinite or very large, and only a portion of it is explored. Usually a *coverage criterion* is specified to provide a way of determining that an adequate sample of behaviors has been inspected. Coverage criteria include path related criteria (used, e.g., in structural coverage schemes) and input/output class criteria (used, e.g., in functional testing).

A coverage criterion may be based on an auxillary, folded model of execution. For instance, data flow testing [RW82, CPRZ85] measure representativeness in terms of coverage of certain control-flow subpaths, thereby relating program execution to paths in a flowgraph model.

Model schema: Some form of program text (such as compiled binary).

Representation of the state space: Only the current state is fully represented during exploration.

Oracle: Testing techniques focusing on exploration of the state space, such as structural and functional coverage schemes, usually leave the oracle unspecified. Assertion-checking schemes are programmable oracles; examples include Anna [LvH85] for checking states and TSL [LHM+87] for checking paths.

Simulation.

Folding: Classes of data are folded in the case of symbolic testing. Additional folding of implementation details may occur in simulations based executable specifications such as Petri nets or PAISley [ZS86]. In these techniques, the state space is considerably simplified by comparison to actual program execution, but may still be too large to exhaustively enumerate.

Sampling: Exploring a sample of the behaviors of an abstract model of execution is clearly a testing technique (and subject to the same optimistic inaccuracy), even when conventional program code is absent as in executable specifications. While the notion of coverage criteria has usually been applied only to testing of actual execution, it is applicable as well to other models of execution. But whereas classical testing selects a sample of the execution space indirectly by selecting test data, abstracted models may allow other ways of choosing a sample. The ATTEST symbolic execution system can be directed to select paths according to a structural coverage criterion [CR81], while executable specification systems typically support random choice in simulation. The Argos system for protocol testing also allows heuristic guidance [Hol87].

1.6 Combining fault detection techniques

Taxonomies are created for practical ends. The conventional taxonomy, as presented in [MH81] and elsewhere, is organized in a manner that makes it very useful for test planning, since a major axis of the taxonomy is the type of documents used by each technique. But the conventional taxonomy is not very useful as a guide to devising new techniques, because the distinction between static and dynamic techniques does not capture enough of the tradeoffs involved in designing such a technique. In this section we argue that the extended taxonomy can provide guidance for devising hybrid analysis techniques and integrated approaches to software fault detection (by highlighting the nature and extent of inaccuracy of each technique potentially employed). We illustrate this by discussing some general directions for integration based on the characteristics in Section 1.5, and showing how some existing approaches fit in this framework.

1.6.1 Combining attributes

Many analysis techniques prescribe some attributes of the classification scheme described here, and leave others unresolved. One may treat the attributes as a sort of check list in putting together a complete technique. That is, a technique has not been completely specified until each attribute (folding, sampling, schemata, representation, and oracle) has been determined.

For instance, one can imagine a symbolic testing technique, using the Anna specification language and processor [LvH85] to provide oracles, and testing to an "all-def-use-paths" data flow coverage criterion. Each of these techniques leaves several attributes unspecified, but together they determine each of the attributes. Symbolic testing determines the model schemata, state folding, and representation of the state space, while the data flow coverage criterion determines the degree of sampling. Instrumentation of a program by the Anna processor provides oracles for a class of specifications. One might adjust any one of these parameters independently, for instance using conventional testing in place of symbolic testing (changing the degree of folding) or using a specification-based or fault-based test adequacy criterion in place of data flow coverage.

1.6.2 Using pessimistic techniques to concentrate optimistic techniques

Pessimistic techniques often fail to distinguish between executable and non-executable paths. If the state space of a pessimistic technique can be related to the state space of an optimistic technique, it may be possible to concentrate the optimistic technique on just those portions of the state space that are reported faulty by the pessimistic technique. For instance, if a class of faults can be ruled out by using a pessimistic technique like flow analysis, then optimistic techniques like testing should concentrate on finding other faults. If symbolic evaluation with loop-cutting assertions is used to show that most of the assertions in a program are satisfied, testing should be concentrated on paths that pass through the assertions that cannot be verified. If a data flow analysis technique reports "may" faults (a possible reference to an uninitialized variable), these could be used in the same way to concentrate testing. Anomalies reported by data flow analysis, such as computation of a value that is never used, might be used to trigger more intensive use of a fault-based technique. One integrated validation methodology that uses pessimistic techniques to concentrate optimistic techniques has been described in detail by Osterweil [Ost84].

1.6.3 Combining pessimistic techniques

Two pessimistic techniques may also benefit from combination, if one is more pessimistic (folds the state space farther) than the other. As mentioned earlier, Chapter 4 of this dissertation proposes combining static concurrency analysis with symbolic execution on this principle. The less pessimistic technique need only follow paths that lead the more pessimistic technique to errors; an error is reported only if both techniques reach it. One can imagine analogous approaches for techniques based on other model schemas. For instance, it is not difficult to show that interpreting a Time Petri net [Mer74] as a standard (untimed) Petri net is a more pessimistic approach than interpreting the time information.[5] That is, the untimed interpretation will reach every state reachable by the time version. If analyzing the standard interpretation is cheaper (because it folds together states that differ only with respect to remaining enable time), it could be used to guide the timed interpretation.

[5]The standard interpretation will contain a sequence of firings for every sequence of firings in the time interpretation. This will make it pessimistic for most classes of errors, but it is possible to compose a branching time temporal logic assertion that may be satisfied only by the standard interpretation, e.g., an assertion that a certain state is reachable but not inevitable.

1.6.4 Combining optimistic techniques

A technique that attempts to raise confidence about the whole space of program behaviors by exploring a portion of the state space depends implicitly on a claim that the portion explored is *representative* of other portions. A coverage criterion identifies classes of states or paths that are similar in some way, so that any member of a class can be taken as a representative of other members of the same class. Confidence hinges on the proposition that if one path in a class is faulty, other paths in the same class are likely to be faulty also, and so exploring one path in the class is likely to uncover a fault. In principle, therefore, one should attempt to identify classes of paths that are treated identically in all important respects. A step toward this ideal is to combine coverage metrics that group paths by different criteria, e.g., functional and structural testing. A technique that combines different coverage criteria in this way is partition analysis.

Partition analysis. Richardson and Clarke have described a hybrid fault detection technique that illustrates some of the principles described above [RC85]. The input domain of a program (and thus, paths through the program state space) are partitioned according to treatment by the specification and, independently, according to treatment by the implementation. The implementation partition is derived by symbolic execution, so that each class of paths in the implementation is represented by a *(path condition, path expression)* pair. The specification partition (classes of inputs treated the same in the specification) is represented similarly as a set of *(condition, computation)* pairs.

The specification partition and implementation partition are combined by considering every combination of classes, i.e., the Cartesian product of the two partitions. For each pair in the combined partition, the respective path conditions are conjoined. If the combined path condition is inconsistent, the class is empty. Otherwise, proof techniques are used to show that the path expression derived from the implementation is equivalent to the computation required by the specification.

Verification of each combined non-empty class of paths is followed by testing. Since proving that path expressions are equivalent to required computations is pessimistic, testing can concentrate on those partitions where proving fails. But in contrast to the strategy described above for combining pessimistic and optimistic techniques, partition analysis calls for testing even when verification is successful. Testing is called for because of potential differences between the implementation of a program on a real machine and the interpretation of path expressions in terms of an ideal, theoretical implementation.

1.7 Summary

The dichotomy between static and dynamic analysis in the conventional taxonomy of fault detection techniques [MH81] is too coarse a distinction to serve as a guide for combining techniques and devising new, hybrid fault detection techniques. We have proposed to partially remedy that situation by replacing the static/dynamic distinction by a distinction between *sampling* the space of possible behaviors, and *folding* states together to make the space smaller. This distinction is better at capturing important tradeoffs in the design of state-space analysis techniques.

All practical techniques are vulnerable to some kind of inaccuracy. The distinction between *pessimistic* inaccuracy (characteristic of techniques that limit effort by folding states together) and *optimistic* inaccuracy (characteristic of techniques that explore only a sample of a state space) is the major dimension of the extended taxonomy. In many cases this distinction coincides with the static/dynamic dichotomy: Most static analysis techniques are pessimistic, and all dynamic analysis techniques are optimistic. The folding/sampling distinction, though, is not just a new name for the old dichotomy. This is shown most clearly by the case of symbolic evaluation, which is conventionally considered a static technique, but which actually encompasses both folding and sampling techniques.

By focusing on analysis tradeoffs, and especially on the relation between strategies for state-space exploration and inaccuracy in the pessimistic or optimistic direction, the revised taxonomy suggests some fruitful areas of research. For instance, coverage criteria are applicable in principle to any sampling technique, and not just to conventional program testing. And while coverage criteria, which relate the extent and nature of sampling to the extent of optimistic inaccuracy, is an active research topic, there is currently no analogous theory characterizing the relation of folding to pessimistic inaccuracy.

Finally, a taxonomy that recognizes the design tradeoffs inherent in devising fault detection techniques is a useful guide to the potential interactions between techniques. This should become increasingly important in the future, as more research moves beyond consideration of individual techniques applied in isolation to integrated application of combinations of techniques and new hybrid techniques.

Chapter 2
Error-preserving abstractions

2.1 Introduction

Chapter 1 classified analysis techniques according to how they sampled or folded a state space, and the resulting optimistic or pessimistic inaccuracy. To make use of such a classification scheme in devising hybrid analysis techniques, we need a fuller characterization of the relations between sampling and optimistic inaccuracy, and between folding and pessimistic inaccuracy. The currently active field of test coverage metrics aims can be viewed as an attempt to characterize the extent of inaccuracy introduced by various strategies for sampling the space of possible program executions. Such a characterization is useful in devising a testing strategy that provides a useful level of assurance for a reasonable investment of effort. An analogous characterization of effort/accuracy tradeoffs for the large class of static analysis techniques based on state-space models of program execution is needed. This chapter provides such a characterization.

Analysis effort in a state-space technique depends critically on leaving out the right details in order to make the space smaller or more regular. In devising or refining an analysis technique, one often wishes to make an argument of the form, *"Ignoring such-and-such a detail may cause some spurious error reports, but it won't cause any errors to go undetected."* In terms of the previous chapter, the claim is that a simplification introduces *only* pessimistic inaccuracy. Such a claim is easy to state informally, but tedious to state precisely and to justify rigorously. In practice, one often settles for a little hand-waving.

We provide a general statement of the property (which we call *error-preserving*) and a set of lemmas that can be used to prove that a particular simplification is error-preserving. We also show that the claim is only meaningful when made with respect to particular classes of specifications. Simplifications that preserve particular errors such as deadlock have been described elsewhere (e.g., the Petri net reductions of Berthelot [BRV80]). The conditions given here are relative to formulas in temporal

logic, which allows them to be used when validating software against user-supplied specifications as well as for detecting a variety of common errors.

The rules developed in this chapter are focused particularly on concurrent software. Although the rules are equally valid for reasoning about models of sequential software, temporal logic specifications are useful primarily when reasoning about concurrency.

Outline of the chapter. Section 2.2 of this chapter gives a precise statement of a property, *error-preserving,* which captures the requirement that transformation of a state-space model not hide errors. Section 2.3 formalizes the notion of "leaving out details" in a state-space model. Section 2.4 gives sufficient conditions for showing that the effect of leaving out a detail is error-preserving with respect to a global safety property expressed in propositional logic. Section 2.5 extends the result to temporal logic specifications. Sections 2.4 and 2.5 also elucidate the relation between details that can be left out and the class of specifications against which programs are to be validated. Section 2.7 demonstrates application of these rules. Section 2.9 concludes.

2.2 Preserving errors

A simplification is a function that maps models to models. A fault detection technique maps models into booleans. Informally, the property we desire is that simplifying a model of execution in some way (by omitting certain details, or making simplifying assumptions) will not cause any faults to go undetected. To formalize this property, it must be made relative to a class of models and to a class of specifications, both of which will be considered in more detail below. (Recall from the Introduction, page 1, that a *fault* is a discrepancy between a specification and an implementation.) Ignoring the finer structure of models and specifications for the moment, the desired property is formalized as follows:

$\mathcal{S}$ is a class of models

$\mathcal{F}$ is a class of specification formulas with a well-defined interpretation such that, for some particular model S and a formula f, either $\mathrm{S} \models f$ (the model *satisfies* the formula) or $\mathrm{S} \not\models f$ (the model *violates* the formula).

$\mu: \mathcal{F} \times \mathcal{S} \rightarrow \{True, False\}$
is a procedure intended to determine whether or not a model (of a particular program) satisfies its specification. Ideally $\mu(f, \mathrm{S})$ would return *True* exactly when $\mathrm{S} \models f$, but practical checking procedures for non-trivial classes of specification formulas will not always have this property.

$\phi: \mathcal{S} \rightarrow \mathcal{S}$
describes the way details are left out in order to make analysis tractable. Intuitively, it seems we ought to speak of two classes of models, $\mathcal{S}$ and $\mathcal{T}$, and describe a simplification as a map from $\mathcal{S}$ to $\mathcal{T}$. However, the formal development is tidier if we deal with one class $\mathcal{S}$ that includes both the domain and range of ϕ, and define ϕ as the identity function when applied to an already-simplified model.

Definition 1 *A map ϕ is* error-preserving *with respect to a formula f iff, for all* $\mathrm{S} \in \mathcal{S}$

$$if\,(\mathrm{S} \not\models f)\ and\ \mu(f, \mathrm{S}) = False\ then\ \mu(f, \phi(\mathrm{S})) = False$$

In words: If a fault would be detected in S, then a fault will be detected in $\phi(\mathrm{S})$ as well.

In this chapter, attention is restricted to propositional logics in which satisfaction ($\models$) can be treated as a total, boolean-valued function.

One might like to reason directly about μ, but this is only possible if certain properties of μ are known. Instead, we reason about actual satisfaction, $\models$. In most cases this will be more useful than reasoning about μ. If μ is not guaranteed to be perfectly accurate, there is no advantage in making sure it makes the same mistakes when a model is simplified. If μ is at least conservative (returns *false* whenever a specification is violated), and ϕ preserves errors, then we can make the following inference:

$$\frac{\begin{array}{ll}\mathrm{S} \not\models f \text{ implies } \mu(f, \mathrm{S}) = False & (\mu \textit{ conservative}) \\ \mathrm{S} \not\models f \text{ implies } \phi(\mathrm{S}) \not\models f & (\phi \textit{ error-preserving})\end{array}}{\mathrm{S} \not\models f \text{ implies } \mu(f, \phi(\mathrm{S})) = False}$$

Besides being more useful than reasoning directly about a checking procedure, reasoning about the actual satisfaction of a formula in a state space (sometimes

called a *structure* or *model*) allows us to use the axioms of a logical system (first propositional logic, and then propositional temporal logic) to derive conditions that guarantee the error-preserving property. Properties of ϕ with respect to a formula f are related to properties of ϕ with respect to component sub-formulas of f. Equivalent reasoning with respect to μ is impossible if μ is allowed to be absolutely arbitrary in its interpretation of formulas.

Development of sufficient conditions for establishing that a simplification function ϕ is error-preserving will require a pair of properties, one for falseness-preservation and one for truth-preservation:

Definition 2 *A map ϕ is* falseness-preserving *with respect to a class of models* S *and a specification formula f, written* $\mathrm{FP}(\phi, f)$, *iff for all* $\mathrm{S} \in \mathcal{S}$, *if* $\mathrm{S} \not\models f$ *then* $\phi(\mathrm{S}) \not\models f$

Definition 3 *A map ϕ is* truth-preserving *with respect to a class of models* S *and a specification formula f, written* $\mathrm{TP}(\phi, f)$, *iff for all* $\mathrm{S} \in \mathcal{S}$, *if* $\mathrm{S} \models f$ *then* $\phi(\mathrm{S}) \models f$

These definitions relate a simplification to a particular specification formula. One usually wants to show that a simplification is error-preserving with respect to a whole class of formulas. Unfortunately, that is not possible for all classes of specification formulas. If a map ϕ is error-preserving with respect to specification formula f, then it will generally not be error-preserving with respect to $\neg f$. For instance, classical techniques for anomaly detection using static data flow analysis [FO76] are conservative with respect to the specification, "definition of variable v always precedes reference to variable v", but would not be conservative with respect to the (rather strange) specification "definition of variable v sometimes does not precede reference to variable v." This is seldom a problem for techniques that detect a fixed class of errors (e.g., deadlocks, uninitialized variables). For analysis techniques that check formulas in a rich specification language, this limitation is an inevitable reflection of the tradeoffs involved in deciding which details to omit from a model.

2.3 Leaving out details

Omitting (or ignoring) some details has the effect of folding many states into a smaller number of states, or of making the structure of a model more regular. To make this notion precise, the structure of a model is given in more detail:

Definition 4 (Model) *A state-space model of program execution,* S, *is a tuple* $\{Q, I, R, \eta\}$,

Q *is a (finite or infinite) set of states.*

$I \subseteq Q$ *is a set of initial states.*

$R \subseteq Q \times Q$ *is a total binary relation*

η: $Q \mapsto 2^{\mathcal{P}}$

where $\mathcal{P}$ *is a set of atomic propositions.*

A variety of similar formalisms are described in the literature. A rationale for choosing this particular definition may be helpful to the reader. Something like function η assigning atomic propositions to states is used in systems for describing structures in which a temporal logic formula is satisfied or not satisfied. Often, such a structure is defined as a set of paths (see, e.g., [Krö87, EH83]) and the underlying directed graph is left implicit. We take the directed graph as primary, since "leaving out details" is something one does with the states (omitting some part of a state vector) or edges (defining a new $R' \supset R$ by ignoring some detail).

$\mathcal{P}$ is not a component of a particular model. When comparing two models (e.g., S and ϕ(S)), it is always assumed that they are defined with respect to the same set $\mathcal{P}$. It makes no sense to speak of one model as an abstraction of another unless it has some of the same properties.

If $\phi(\{Q, I, R, \eta\}) = \{Q', I', R', \eta'\}$, conditions necessary to ensure that ϕ is error-preserving with respect to a particular specification formula will be stated as conditions on $\phi_Q: Q \rightarrow Q'$. Whereas ϕ maps models to models, ϕ_Q associates states in a particular model (before simplification) with states in the simplified version. For the class of functions ϕ that correspond to the intuitive notion of "leaving out details," ϕ_Q is primary and ϕ results from replacing each element q of Q with $\phi_Q(q)$. In the remainder of the chapter, subscripts on ϕ will be omitted where the particular function is clear from context.

Requiring ϕ_Q to be a (total) function means that every state in the original model is mapped to exactly one state in the 'simplified' model. The mapping need not be one-to-one or onto. Consider a model in which each state is identified by a tuple of attributes and a simplification that consists of omitting one of the attributes. Omitting an attribute collapses the state space along one dimension. Since that attribute may partially determine R in the original model, simplification may also introduce new states that were not reachable in the original. For example, graph-based analysis such as data flow analysis, in which data values are omitted, compresses the infinite space of actual execution states into a finite space of program locations, but also introduces unexecutable paths.

The models discussed here are operational in flavor, with explicit states and transitions. This approach is compared to an alternative, denotational framework in Section 2.8.

2.4 Global safety properties of states

We begin by considering specifications that describe individual states, where specifications are expressed in propositional logic quantified over all states in a model. These specification formulas are implicitly or explicitly prefixed by $\forall q \in Q$. A program is correct if *every* state satisfies the specification formula.

Satisfaction of a propositional formula in a state is defined as one would expect: Formula f is evaluated by replacing each propositional variable by *True* if it belongs to $\eta(q)$ and *False* otherwise. The definitions of truth-preserving and falseness-preserving are extended to apply to ϕ_Q in the obvious way. ϕ is falseness-preserving (respectively, truth-preserving) for a class of models if ϕ_Q is falseness-preserving (truth-preserving) for all states.

The following conditions are sufficient to show that ϕ applied to model S is a falseness-preserving abstraction with respect to formula f, where f is interpreted as a global safety property of states.

F1. if f is an atomic proposition, and $f \notin \eta(q)$, then $f \notin \eta(\phi(q))$
F2. if f is a conjunction $a \wedge b$, then $\mathrm{FP}(\phi, a)$ and $\mathrm{FP}(\phi, b)$
F3. if f is a disjunction $a \vee b$, then $\mathrm{FP}(\phi, a)$ and $\mathrm{FP}(\phi, b)$
F4. if f is a negated formula $\neg a$, then $\mathrm{TP}(\phi, a)$

Condition F4 calls for a corresponding set of conditions sufficient to guarantee that ϕ is truth-preserving with respect to some component sub-formula.

T1. if f is an atomic proposition, and $f \in \eta(q)$, then $f \in \eta(\phi(q))$
T2. if f is a conjunction $a \wedge b$, then $\mathrm{TP}(\phi, a)$ and $\mathrm{TP}(\phi, b)$
T3. if f is a disjunction $a \vee b$, then $\mathrm{TP}(\phi, a)$ and $\mathrm{TP}(\phi, b)$
T4. if f is a negated formula $\neg a$, then $\mathrm{FP}(\phi, a)$

Condition F1 is the base case of a recursive consideration of formulas. Together with F2 and F3, F1 determines a set of propositional variables that must be "biased false" in mapping states. That is, when states $q_1, q_2, q_3 \ldots$ are mapped to the same state $\phi(q)$, the proper value of a proposition a identified by rule F1 is $\eta(q_1) \wedge \eta(q_2) \wedge \eta(q_3) \wedge \ldots$

As F1–F3 determine a set of variables that must be combined with a logical *AND* operation when states are merged, T1–T3 determine a set of variables that must be combined with logical *OR*. F4 and T4 involve mutual recursion between the two sets of conditions. Since rules T2–T4 and F2–F4 are defined in terms of smaller formulas at each step, applying them will eventually terminate with a set of variables identified by F1 and another set identified by T1. If a variable falls into both sets, then it must be shown that ϕ does not merge states with different values of that variable.

Conditions F1–F4 and T1–T4 can be restated as lemmas. Equivalent lemmas for temporal logic formulas, which subsume the propositional logic formulas considered here, are stated and proved in Section 2.5.3. It is easy to show that these rules are too strong to be necessary conditions. Applied to the specification formula $a \vee \neg a$, these rules will prohibit ϕ from merging states with different values of a, even though every function ϕ is error-preserving with respect to this degenerate specification.

A small example. Suppose a technique were required to check a state-space model against a specification stated as $(a \vee b) \rightarrow (b \wedge c)$. A simplification is proposed that will have the effect of folding together states that differ in some details. The rules given here tell how properties a, b, and c must be treated in merged states. Where states are merged with differing values of a, they must be combined with logical *OR* (to be truth-preserving). Where states are merged with differing values of c, they must be combined with *AND* (to be falseness-preserving). States with differing values of b cannot be merged, since b must be both truth-preserved (because it appears in the premise of the specification, i.e., negated) and falseness-preserved (because it appears in the conclusion).

Consider parallel action analysis using Taylor's technique for analyzing tasking Ada programs [Tay83b]. An (oversimplified) property to be checked might be "task T never modifies variable v at the same time another task is reading v." The program flowgraph can be marked with propositional variable w to indicate the states in which task T writes v, and marked with r to indicate states in which some task reads v. The analysis technique uses reduced flowgraphs, in which flowgraph nodes not involved in task interaction are merged together. The rules given above comport with the intuitive requirements for reducing the flowgraph when the spec $\neg(w \wedge r)$ is to be checked: a node in the reduced flowgraph must be marked with w (or r) if any of the nodes it represents are marked with w (or r). This requirement hardly needs justification for the case of read/write conflicts — the advantage of using rules T1–T4 and F1–F4 above is that the flowgraph reduction procedure could be similarly guided for arbitrary, user-specified parallel action analysis.

Properties of reachable states. Many practical analysis techniques use relation R to explore a state space. That is, R is represented as a function for generating new states from initial states and states already generated. In this case, the specification formula f is implicitly quantified over reachable states. If the set of states Q may contain unreachable as well as reachable states, the following conditions will ensure that ϕ maps reachable states onto reachable states:

Definition 5 *A map ϕ is* connectivity-preserving *iff*

(Roots) *if $i \in I$, then $\phi(i) \in I'$, where I and I' are the sets of initial states in* S *and ϕ*(S)*, respectively.*

(Edges) *if $q_1 R\, q_2$ and $\phi(q_1) \neq \phi(q_2)$, then $\phi(q_1) R\, \phi(q_2)$*

The first condition requires initial states to map to initial states. The second condition preserves connections between states, except that self-loops need not be introduced when connected states are merged. These two conditions, together with F1–F4 and T1–T4, are sufficient to show that a model simplification ϕ is error-preserving with respect to a propositional formula f that is implicitly quantified over reachable states.

2.5 Temporal properties

A number of systems for specifying sequence constraints (allowed and disallowed sequences of events or operations) have been proposed. Several of these are based on formal languages, typically regular expressions [KS83, OO86, Mac82]. Recently, more interest has been shown in temporal logics for specifying allowable sequencing [Lam83] and systems for checking software against temporal logic specifications have begun to appear [FRV85, CES86, MR87].

The previous section established conditions under which details could be omitted from a state-space model of program execution without causing faults to go undetected. The conditions were relative to particular specification formulas stated in propositional logic, and interpreted as applying globally to all program states. In this section we derive conditions that allow leaving out details when the specification to be checked is a formula of propositional temporal logic. Section 2.5.1 introduces the system of temporal logic, Section 2.5.2 gives general properties ϕ should exhibit, and Section 2.5.3 derives sufficient conditions to show that ϕ is an error-preserving abstraction with respect to a formula in temporal logic, provided the general properties are obeyed.

2.5.1 Background: Temporal logics

A temporal logic is a standard (propositional or first-order) logic augmented with temporal, or time-oriented, operators. Typical operators are "eventually," "always," and "until." A temporal logic formed by adding such operators to a propositional logic is a propositional temporal logic.

The major families of temporal logics are the linear time logics and the branching time logics. (An extension to these, called interval logic, is not considered here.) A linear time logic describes properties along a particular path; implicitly, a software specification employing linear time logic is universally quantified with respect to program paths. A branching time logic allows consideration of alternative paths, and usually allows both existential and universal quantification over paths. A branching time logic formula describes a state (so that quantification is over paths leaving that state) and a linear time formula describes a path. Practical analysis techniques have so far been devised only for restricted branching-time logics, and checking a finite state-space model for compliance with an unrestricted linear time specification is known to be an NP-complete problem [SC85].[1]

It is possible to combine linear and branching time logics in a single system. The CTL* logic of Emerson and Halpern [EH83] (see Table 2.1) defines both *state formulas* and *path formulas*, by separating the temporal operators **F** (future), **G** (global), **U** (until), and **X** (next state) from the path quantifiers **A** (all paths) and **E** (some paths). A linear time formula like $a \rightarrow \Diamond b$ (in the conventional notation) is expressed as $\mathbf{A}(a \rightarrow \mathbf{F}b)$, where the **A** quantifier makes explicit the fact that a linear time specification applies to *all* possible program executions.

2.5.2 General conditions on ϕ

Rules for showing that a simplification function ϕ is error-preserving with respect to a propositional logic specification were based on a function ϕ_Q from states to states. Analogous rules for temporal logic specifications must be based on a function ϕ_X from sequences to sequences. This section provides the necessary definitions, and sets down general properties that such a function must have. Section 2.5.3 then provides rules for showing that such a function is error-preserving with respect to a specific temporal logic specification.

[1] Model-checking for linear time logics is exponential in the length of the formula, rather than the size of the model (state space), so conceivably an acceptably efficient analysis technique might be based on a linear time temporal logic [LP85]. It has also been shown that an efficient algorithm for checking linear time formulas can be extended to check formulas in a combined linear and branching time logic for essentially no cost in efficiency [EL85].

Table 2.1: Syntax and semantics of a temporal logic combining branching time and linear time formulas (adapted from [EH83]). $\langle \text{sf} \rangle$ is a state formula (branching time logic formula), $\langle \text{pf} \rangle$ is a path formula (linear time logic formula).

Syntax	*Interpretation*
$\langle \text{sf} \rangle ::= \textit{variable}$	$q \models a$ iff $a \in \eta(q)$
$\langle \text{sf} \rangle ::= \langle \text{sf} \rangle \wedge \langle \text{sf} \rangle$	$q \models a \wedge b$ iff $(q \models a)$ and $(q \models b)$
$\langle \text{sf} \rangle ::= \neg \langle \text{sf} \rangle$	$q \models \neg a$ iff $q \not\models a$
$\langle \text{sf} \rangle ::= \mathbf{A} \langle \text{pf} \rangle$	$q \models \mathbf{A}a$ iff $\forall_{x \in X(\text{S})}((\text{head}(x) = q) \Rightarrow x \models a)$
$\langle \text{sf} \rangle ::= \mathbf{E} \langle \text{pf} \rangle$	$q \models \mathbf{E}a$ iff $\exists_{x \in X(\text{S})}((\text{head}(x) = q)$ and $x \models a)$
$\langle \text{pf} \rangle ::= \textit{variable}$	$x \models a$ iff $a \in \eta(\text{head}(x))$
$\langle \text{pf} \rangle ::= \langle \text{pf} \rangle \wedge \langle \text{pf} \rangle$	$x \models a \wedge b$ iff $(x \models a)$ and $(x \models b)$
$\langle \text{pf} \rangle ::= \neg \langle \text{pf} \rangle$	$x \models \neg a$ iff $x \not\models a$
$\langle \text{pf} \rangle ::= \mathbf{G} \langle \text{sf} \rangle$	$x \models \mathbf{G}a$ iff $\forall_{i \geq 0}(\text{head}(\text{tail}(i, x)) \models a)$
$\langle \text{pf} \rangle ::= \mathbf{F} \langle \text{sf} \rangle$	$x \models \mathbf{F}a$ iff $\exists_{i \geq 0}(\text{head}(\text{tail}(i, x)) \models a)$
$\langle \text{pf} \rangle ::= \mathbf{G} \langle \text{pf} \rangle$	$x \models \mathbf{G}a$ iff $\forall_{i \geq 0}(\text{tail}(i, x) \models a)$
$\langle \text{pf} \rangle ::= \mathbf{F} \langle \text{pf} \rangle$	$x \models \mathbf{F}a$ iff $\exists_{i \geq 0}(\text{tail}(i, x) \models a)$
$\langle \text{pf} \rangle ::= \mathbf{X} \langle \text{sf} \rangle$	$x \models \mathbf{X}a$ iff $\text{head}(\text{tail}(1, x)) \models a$
$\langle \text{pf} \rangle ::= \langle \text{sf} \rangle \mathbf{U} \langle \text{sf} \rangle$	$x \models a\mathbf{U}b$ iff $\exists_{i \geq 0}(\text{head}(\text{tail}(i, x)) \models b$ and $\forall_{j \geq 0}((j \leq i) \Rightarrow \text{head}(\text{tail}(j, x)) \models a))$
$\langle \text{pf} \rangle ::= \mathbf{X} \langle \text{pf} \rangle$	$x \models \mathbf{X}a$ iff $\text{tail}(1, x) \models a$
$\langle \text{pf} \rangle ::= \langle \text{pf} \rangle \mathbf{U} \langle \text{pf} \rangle$	$x \models a\mathbf{U}b$ iff $\exists_{i \geq 0}(\text{tail}(i, x) \models b$ and $\forall_{j \geq 0}((j \leq i) \Rightarrow \text{tail}(j, x) \models a))$

Paths. *Paths* are defined in terms of the relation R, which for simplicity is assumed to be total. (Terminal states can be treated by making each terminal state its own successor. Deadlock must be treated as a property of a state; see Section 3.3.3 of Chapter 3 for details.) A path is an infinite sequence of states, $q_0, q_1, q_2, \ldots$ such that $q_i R q_{i+1}$ for all $i \geq 0$. The set of all such paths in a particular model is denoted $X(\mathrm{S})$, and a particular path will be denoted x. The tail of a path will be denoted $\mathrm{tail}(x)$, and a tail of x beginning with the n^{th} element will be denoted $\mathrm{tail}(n, x)$ ($\mathrm{tail}(0, x) = x$, and $\mathrm{tail}(1, x) = \mathrm{tail}(x)$). The first state on path x is denoted $\mathrm{head}(x)$. The catenation of state q with path x is denoted $q \cdot x$.

To reason about the effects of simplifying a model that is to be checked against sequence specifications, a mapping from paths in S to paths in $\phi(\mathrm{S})$ is required. A pointwise extension of ϕ_Q, i.e.

$$\phi_X(q_0, q_1, q_2, \ldots) = \phi_Q(q_0), \phi_Q(q_1), \phi_Q(q_2), \ldots$$

will not do the trick, because it may not produce valid paths in $\phi(\mathrm{S})$. If q_1 and q_2 map to the same state $\phi(q)$, and $q_1 R q_2$, then subsequences $\ldots q_1 q_2 \ldots$ will map onto subsequences $\ldots \phi(q)\phi(q) \ldots$, and it may not be the case that $\phi(q) R \phi(q)$. We need a mapping that "compresses out" sequences of states that have been merged. First, another constraint on ϕ is needed:

Definition 6 *We say ϕ is* loop-preserving *if $q_1 R q_2 R q_3 \ldots q_n R q_1$, and $\phi(q_1) = \phi(q_2) = \phi(q_3) \ldots = \phi(q_n)$ implies $\phi(q_1) R \phi(q_1)$.*

The loop-preserving property guarantees that any loop in $X(\mathrm{S})$ will be reflected in a loop in $X(\phi(\mathrm{S}))$, provided Q is finite. Without it, "compressing out" merged subsequences could cause an infinite path to map to a finite sequence, which is not a path. This corresponds to merging nodes in a way that hides an infinite loop in program execution (see Figure 2.1).

With the restriction on loop preservation, the following definition of a mapping ϕ_X will be satisfactory for finite state spaces:

Definition 7 *The* standard extension *of ϕ_Q to ϕ_X is defined by*

$$\phi_X(x) = \phi_Q(\mathrm{head}(x)) \cdot \phi'(\mathrm{head}(x), \mathrm{tail}(x))$$

$$\phi'(q, x) = \begin{cases} \phi'(q, \mathrm{tail}(x)) & \text{if } \phi_Q(q) = \phi_Q(\mathrm{head}(x)) \\ & \quad \wedge \phi_Q(q) \not R \phi_Q(q) \\ \phi_X(x) & \text{otherwise} \end{cases}$$

An additional complication arises in infinite state spaces: an infinite sequence of states may never repeat a state. An example is a non-terminating program loop

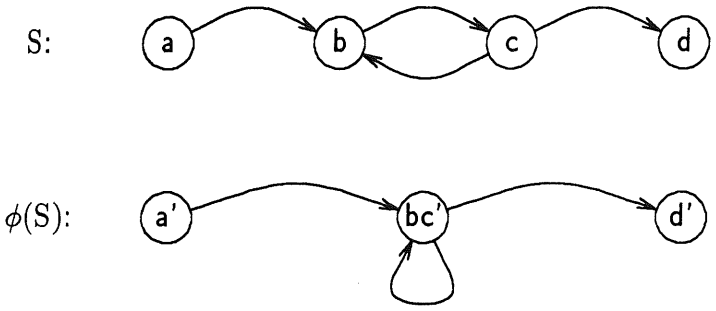


Figure 2.1: Loops must be preserved. If the loop between states b and c in S were not reflected by a self-loop on state bc' in ϕ(S), formula **AF***d*, which is false in S, would become true in ϕ(S).

in which a variable takes a new value on each iteration. In such a case, ϕ_X as defined above may not be a function from infinite sequences to infinite sequences.

It is difficult to define a function ϕ_X that is always satisfactory for infinite state spaces, but satisfactory functions are usually not hard to define for a particular simplification. The conditions ϕ_X should satisfy (i.e., the properties we will use in the proofs below) are:

Definition 8 *ϕ is* order-preserving *iff for all x, for all $i \geq 0$,*

1. $\phi(\text{head}(x)) = \text{head}(\phi(x))$
2. $\exists_{j \geq 0}\ (\phi(\text{tail}(i, x)) = \text{tail}(j, \phi(x)))$
   (Informally: Every tail of x is represented).
3. $\exists_{j \geq 0}\ (\text{tail}(i, \phi(x)) = \phi(\text{tail}(j, x)))$
   (Informally: Tail of image is image of some tail).

It can be shown that the particular function ϕ_X defined above for finite state spaces meets these conditions. Condition 2 implies that ϕ_X preserves the order of states on a path in the following sense:

If ϕ_X satisfies condition 2 above, then, if x is a path and $j \geq i$, then
$\exists_{0 \leq i' \leq j'}\ (\phi(\text{tail}(i, x)) = \text{tail}(i', \phi(x))$ *and* $\phi(\text{tail}(j, x)) = \text{tail}(j', \phi(x)))$

Proof: $\text{tail}(j, x)$ is $\text{tail}(i + k, x)$ for some non-negative k, which may be rewritten as $\text{tail}(k, \text{tail}(i, x))$. $\phi(\text{tail}(k, \text{tail}(i, x)))$ is $\text{tail}(j', \phi(\text{tail}(i, x)))$, by condition 2 above.

An analogous but converse condition can be proven from condition 3; hence they are together called the *order-preserving* property.

Fairness assumptions. Loop preservation is not always appropriate. Some extant static analysis methods fail to guarantee loop preservation, with the result that they may fail to detect some infinite loops. For instance, Taylor's static analysis technique for tasking Ada programs may reduce a program loop between synchronization points to a single flowgraph node. A synchronization error resulting from an infinite loop may go undetected by the technique. In fact, reporting such errors would be silly and counterproductive, since sequential program logic is not the focus of the technique. A similar situation arises in data flow analysis: data flow analysis does not report a possible def-without-use anomaly after loops, just because the loop might be infinite.

An appropriate way to deal with these techniques is to consider that they make a fairness assumption, and ϕ_X is restricted to paths that obey that assumption. The most common fairness assumption is just that all loops (or all loops collapsed by leaving out details) terminate. Under this assumption, loop preservation is not necessary, and the function ϕ_X defined by the standard extension of ϕ_Q will work for infinite as well as finite state spaces.

2.5.3 Conditions relative to specification formulas

Given a mapping ϕ_X as described above, the definition of error-preserving abstraction is extended naturally to state formulas and path formulas in propositional temporal logic. The following properties can be derived from the definitions in Table 2.1. (The properties are stated here as lemmas, but can also be stated in the form of a recursive procedure for obtaining the sets of atomic formulas to be preserved, as was done for global safety properties of states.) For the remainder of this section, it is assumed that ϕ_X is loop-preserving and order-preserving as defined above.

Lemma 1 *If a and b are state formulas, and* $\mathrm{FP}(\phi, a)$ *and* $\mathrm{FP}(\phi, b)$*, then* $\mathrm{FP}(\phi, a \wedge b)$ *and* $\mathrm{FP}(\phi, a \vee b)$*. If a and b are path formulas, and* $\mathrm{FP}(\phi, a)$ *and* $\mathrm{FP}(\phi, b)$*, then* $\mathrm{FP}(\phi, a \wedge b)$ *and* $\mathrm{FP}(\phi, a \vee b)$*.*

Proof: (State formula version)
Suppose $s \not\models (a \wedge b)$, for s a state in S. Wlog, assume $s \not\models a$. From $\mathrm{FP}(\phi, a)$ we have $\phi(s) \not\models a$, and therefore $\phi(s) \not\models (a \wedge b)$.
Suppose $s \not\models (a \vee b)$, for s a state in S. Then $s \not\models a$ and $s \not\models b$. From $\mathrm{FP}(\phi, a)$ and $\mathrm{FP}(\phi, b)$, we have $\phi(s) \not\models a$ and $\phi(s) \not\models b$, and therefore $\phi(s) \not\models (a \vee b)$.
The path formula version is identical, with path x substituted for state s.

Lemma 2 *If a is a path formula or state formula, and if* $\mathrm{FP}(\phi, a)$*, then* $\mathrm{TP}(\phi, \neg a)$*. If a is a path formula or state formula, and if* $\mathrm{TP}(\phi, a)$*, then* $\mathrm{FP}(\phi, \neg a)$*.*

Proof: Suppose $x \models \neg a$, and $\mathrm{FP}(\phi, a)$ for a path formula a. Then $x \not\models a$, and therefore $\phi(x) \not\models a$. Therefore $\phi(x) \models \neg a$.
Proofs of the remaining cases are similar.

Lemma 3 *If a is a path formula or state formula, and if* $\mathrm{FP}(\phi, a)$, *then* $\mathrm{FP}(\phi, \mathbf{G}a)$ *and* $\mathrm{FP}(\phi, \mathbf{F}a)$.
If a is a path formula or state formula, and if $\mathrm{TP}(\phi, a)$, *then* $\mathrm{TP}(\phi, \mathbf{G}a)$ *and* $\mathrm{TP}(\phi, \mathbf{F}a)$.

Proof: Assume $\mathrm{FP}(\phi, a)$ for a path formula a, and suppose $x \not\models a$. Then $\neg\exists_{i\geq 0}\,(\mathrm{tail}(i, x) \models a)$, or equivalently, $\forall_{i\geq 0}\,(\mathrm{tail}(i, x) \not\models a)$. From $\mathrm{FP}(\phi, a)$ we conclude $\forall_{i\geq 0}\,(\phi(\mathrm{tail}(i, x)) \not\models a)$, and from the condition on ϕ_X that every tail of an image of x is the image of a tail of x, we have $\forall_{i\geq 0}\,(\mathrm{tail}(i, \phi(x)) \not\models a)$. Thus we obtain $\phi(x) \not\models a$.
The other cases are similar.

Lemma 4 *If a and b are both path formulas, and if* $\mathrm{FP}(\phi, a)$ *and* $\mathrm{FP}(\phi, b)$, *then* $\mathrm{FP}(\phi, a\mathbf{U}b)$.
If a and b are both state formulas, and if $\mathrm{FP}(\phi, a)$ *and* $\mathrm{FP}(\phi, b)$, *then* $\mathrm{FP}(\phi, a\mathbf{U}b)$.

Proof: Assume $\mathrm{FP}(\phi, a)$ and $\mathrm{FP}(\phi, b)$ and $x \not\models a\mathbf{U}b$. Either $\neg\exists_{i\geq 0}\,(\mathrm{tail}(i, x) \models b)$, or for all such i, $\exists_{j, 0\leq j\leq i}\,(\mathrm{tail}(j, x) \not\models a)$.
Case 1: $\neg\exists_{i\geq 0}\,(\mathrm{tail}(i, x) \models b)$. This is identical to inferring $\mathrm{FP}(\phi, \mathbf{F}b)$ from $\mathrm{FP}(\phi, b)$, in the previous lemma.
Case 2: $\exists_{i\geq 0}\,(\mathrm{tail}(i, x) \models b)$, and for all such i, $\exists_{j, 0\leq j\leq i}\,(\mathrm{tail}(j, x) \not\models a)$. Essentially, we need to show that ϕ preserves the order of states at least to the extent that the violation of a still precedes the first suffix satisfying b. Choose an i and j according to the hypothesis of the case; then, there must be a i' and j' such that $\phi(\mathrm{tail}(i, x)) = \mathrm{tail}(i', \phi(x))$ and $\phi(\mathrm{tail}(j, x)) = \mathrm{tail}(j', \phi(x))$. Moreover, $\mathrm{tail}(i, x) = \mathrm{tail}(k, \mathrm{tail}(j, x))$ for some $k \geq 0$. Thus, $j' \leq i'$, and since from $\mathrm{FP}(\phi, a)$ we have $\mathrm{tail}(j', \phi(x)) \not\models a$, we conclude $\phi(x) \not\models a$.

Lemma 5 *If a and b are path formulas, and if* $\mathrm{TP}(\phi, a)$ *and* $\mathrm{TP}(\phi, b)$ *then* $\mathrm{TP}(\phi, a\mathbf{U}b)$.
If a and b are state formulas, and if $\mathrm{TP}(\phi, a)$ *and* $\mathrm{TP}(\phi, b)$ *then* $\mathrm{TP}(\phi, a\mathbf{U}b)$.

Proof: Similar to the version for FP, but use condition 3 of the order-preserving property.

Lemma 6 *If a is a state formula, and if* $\mathrm{FP}(\phi, a)$, *then* $\mathrm{FP}(\mathbf{A}a)$.

Proof: The **A** operator involves universal quantification over paths from a state. If any such path violates a, then the quantified formula must be violated.

Lemma 7 *If a is a state formula, and if* $\mathrm{TP}(\phi, a)$, *then* $\mathrm{TP}(\mathbf{E}a)$.

Proof: The **E** operator involves existential quantification over paths. If any path from a state satisfies a, then existential quantification over such paths must be satisfied.

Additional rules

All of the conditions given here are sufficient but not necessary. Sometimes it is helpful to derive less general rules, tailored to specific simplifications of models. An example of such a rule in analysis of concurrent programs is *virtual coarsening* [Pnu86], which allows a sequence of steps between process interactions to be treated as a single step.

We illustrate with a special rule for inferring FP($\mathbf{F}a$) when merging regions of code. The rule given above requires FP(a). Thus, if $p \not\models a$ but $q \models a$, and $\phi(p) = \phi(q)$ (that is, p and q are combined in the simplified model), then $\phi(p) \not\models a$. This leads to inaccurate models when a sequence of states, the last of which satisfies a, is merged into a single state. (An example where such merging is useful is described below, in Section 2.7.) Provided there is no way to leave the merged region without a becoming true at some point, it is permissible to label the state representing the merged region with a.

Lemma 8 *If* $p \not\models \mathbf{AF}a$ *implies* $\phi(p) \not\models a$, *then* FP($\phi, \mathbf{F}a$).

Proof: Suppose the contrary, that the premise of the lemma is true on some path $x \not\models \mathbf{F}a$ and $\phi(x) \models a$. Then there is no state in x satisfying a, but there is a state in $\phi(x)$ satisfying a. Every state in $\phi(x)$ is the image of some state in x (by the order-preserving property), so there must be a state $p \not\models a$ such that $\phi(p) \models a$. By the premise of the lemma, $p \models \mathbf{AF}a$. Thus $\mathbf{F}a$ must be true of any path passing through p in the original model, contradicting the hypothesis that $x \not\models \mathbf{F}a$.

The use of this lemma in merging sequences of states is illustrated in Figure 2.2.

Missing rules

The **X** operator does not appear in any of the rules above. It must be disallowed because it allows specification of behavior at a level of detail that makes simplification of models impractical. For instance, with the **X** operator, it is easy to construct a specification that prescribes the twelfth state of the system.

The two quantifiers over paths are asymmetric. There is no rule for inferring TP($\phi, \mathbf{A}a$), and no rule for inferring FP($\phi, \mathbf{E}a$). An error-preserving abstraction must not introduce new paths satisfying a formula f if it is to be falseness preserving with

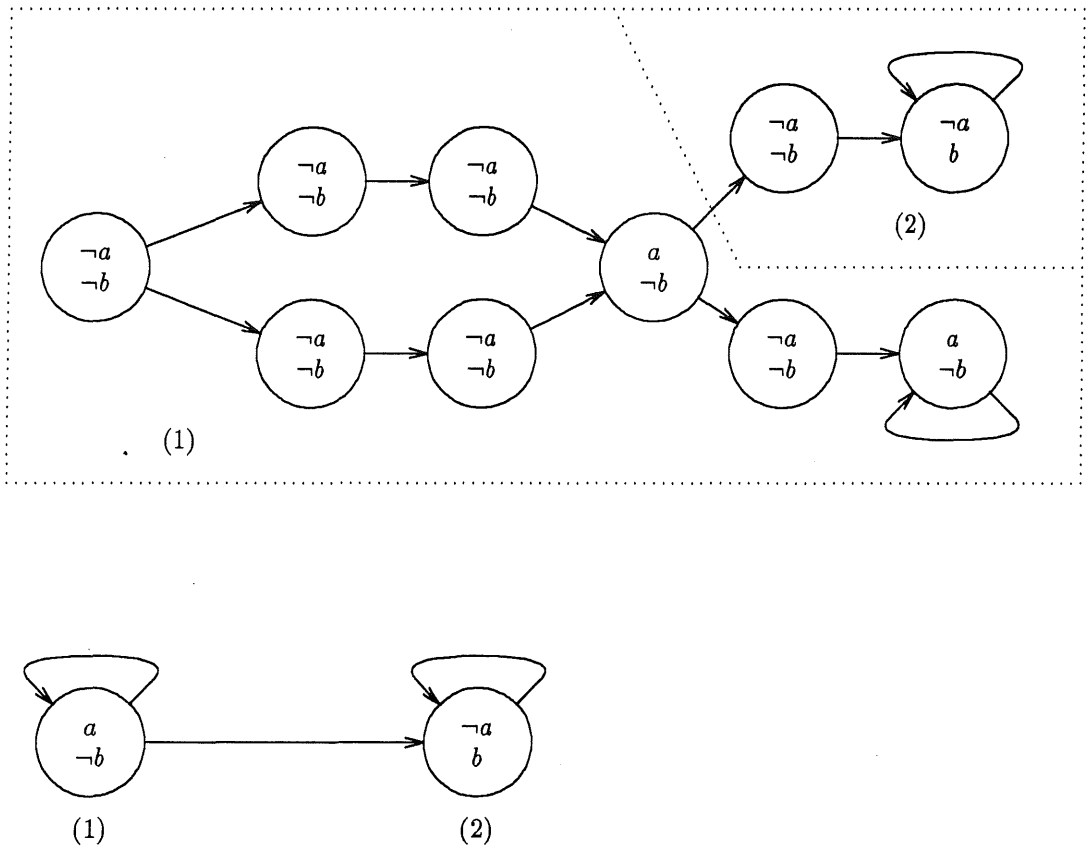


Figure 2.2: Using the special coarsening rule for FP($\mathbf{F}a$). The upper model is simplified into the lower model by merging states. It is permissible to label the representatives of region (1) with a and region (2) with b, even though the falseness of neither proposition is maintained, provided the specification calls only for the *eventuality* of each condition.

respect to formula $\mathbf{E}f$, but it must not omit paths satisfying f if it is to be falseness preserving with respect to formula $\mathbf{A}f$. The conjunction of these two conditions is too strong to allow useful model simplifications.

A system could be defined in which $\mathrm{TP}(\phi, \mathbf{A}a)$ and $\mathrm{FP}(\phi, \mathbf{E}a)$ are both derivable, but $\mathrm{TP}(\phi, \mathbf{E}a)$ and $\mathrm{FP}(\phi, \mathbf{A}a)$ are not. Such a system would be less useful than the one given here, because practical analysis techniques must deal with unexecutable program paths (which would be disallowed under the alternative system). Also, $\mathrm{FP}(\phi, \mathbf{A}a)$ arises naturally when attempting to show a technique is error-preserving with respect to linear time temporal logic specifications, whereas the other forms are much less useful in practice.

Summary of Rules for Temporal Logic

General properties that ϕ should exhibit to be an error-preserving abstraction are

1. ϕ_Q must be a total function from states to states, and ϕ_X must be a total function from paths to paths.
2. ϕ must be connection-preserving (Definition 5).
3. ϕ must be loop-preserving (Definition 6), unless a fairness assumption excludes infinite loops among merged states as discussed in Section 2.5.2.
4. ϕ must be order-preserving (Definition 8).

If ϕ meets each of those conditions, then the lemmas given in this section (which subsume rules F1-F4 and T1-T4 given in Section 2.4) can be applied to show that ϕ is an error-preserving abstraction with respect to a particular temporal logic formula, viz.,

1. If $\mathrm{FP}(\phi, a)$ and $\mathrm{FP}(\phi, b)$, then $\mathrm{FP}(\phi, a \wedge b)$ and $\mathrm{FP}(\phi, a \vee b)$.
2. If $\mathrm{FP}(\phi, a)$, then $\mathrm{TP}(\phi, \neg a)$.
3. If $\mathrm{FP}(\phi, a)$, then $\mathrm{FP}(\phi, \mathbf{G}a)$ and $\mathrm{FP}(\phi, \mathbf{F}a)$.
4. If $\mathrm{FP}(\phi, a)$ and $\mathrm{FP}(\phi, b)$ then $\mathrm{FP}(\phi, a\mathbf{U}b)$.
5. If $\mathrm{FP}(\phi, a)$, then $\mathrm{FP}(\mathbf{A}a)$.
6. If $\mathrm{TP}(\phi, a)$ and $\mathrm{TP}(\phi, b)$, then $\mathrm{TP}(\phi, a \wedge b)$ and $\mathrm{TP}(\phi, a \vee b)$.
7. If $\mathrm{TP}(\phi, a)$, then $\mathrm{FP}(\phi, \neg a)$.
8. If $\mathrm{TP}(\phi, a)$, then $\mathrm{TP}(\phi, \mathbf{G}a)$ and $\mathrm{TP}(\phi, \mathbf{F}a)$.

9. If TP(ϕ, a) and TP(ϕ, b) then TP($\phi, a\mathbf{U}b$).
10. If TP(ϕ, a), then TP($\mathbf{E}a$).

2.6 Sequences of events

The temporal logic considered in the previous section considers only properties of states and sequences of states. Sometimes it is more convenient to consider the events that take a system from one state to the next.

Since it is natural to think of sequences of events as the language generated by the state space considered as an automaton, and since language and automata theory are relatively well-explored research areas, it is not surprising that many analysis techniques for verifying correct sequencing of events are based on automata theory rather than temporal logic. These include the constrained expressions analysis approach of Avrunin et al. [ADWR86, DAW88], and AQRE (*A*nchored *Q*uantified *R*egular *E*xpressions) analysis technique of Olender [OO86]. It is also possible, however, to consider sequences of events in a temporal logic framework, as shown for example by Barringer et. al. [BKP84]. Here we consider temporal logic specifications that may encompass both states and events in the same specification formula by adding event formulas to CTL*.

We introduce two new aspects of a state space model:

Σ is a finite set of operation symbols. Like P, we assume a single Σ when comparing two models.

δ: $Q \times \Sigma \mapsto 2^Q$
is a transition relation, with the proviso that $\delta(q, \sigma)$ must be finite for any $q \in Q$ and $\sigma \in \Sigma$.

The state space models considered thus far have had a transition relation R. R determines which states may follow each other in a computation, but doesn't say what action causes the transition. δ gives information about the event that takes the system from one state to the next as well. Rather than replacing R by δ in the general requirements for error-preserving abstractions (Section 2.5.2), it will be sufficient to treat R as being a derivative property based on δ:

Definition 9 $(p, q) \in R$ *iff* $((p, \sigma), q) \in \delta$, *for some* $\sigma \in \Sigma$.

The definition of paths is changed in the obvious way. Instead of a sequence of states

$$q_0, q_1, q_2, \ldots$$

we have an alternating sequence

$$q_0, \sigma_0, q_1, \sigma_1, q_2, \sigma_2, \ldots$$

where $q_{i+1} \in \delta(q_i, \sigma_i)$. head$(x)$ and tail(n, x) are defined as before.

As the function η gives values of atomic propositions describing states, we require an interpretation function for atomic propositions describing events. η associates the possibly infinite state space with a finite set of atomic propositions. One could imagine a system in which Σ is also allowed to be infinite, but this would complicate matters and is not particularly useful for our current purposes. We take the short cut of using symbols from Σ as patterns to match the corresponding event symbols in a path.

A single rule suffices to treat any event formula as a path formula in temporal logic. A path formula may be an event symbol ($\langle \text{pf} \rangle ::= \langle \text{ef} \rangle$), with the semantics that such a path formula is satisfied if the first event on the path matches the event symbol:

$$q_0, \sigma_0, q_1, \sigma_1, \ldots \models a \text{ iff } \sigma_0 = a$$

The remaining rules from Table 2.1, page 41, remain unchanged. In particular, note that boolean combinations of event formulas are interpreted in the expected way, as if they denoted subsets of Σ, though in fact the rules governing boolean combinations of path formulas are used instead.

Interpretations of the temporal operators using the rules above are consistent with the way events are treated in the temporal logic of Barringer [BKP84]. We note, however, that this semantics (and that of [BKP84]) interprets $a\mathbf{U}e$ in a non-intuitive way: If a is initially false, but e matches the first event on a path, the formula will be satisfied although intuitively one would expect it to be violated. We speculate that the best way to avoid this anomaly is to treat all paths as beginning with events rather than states, and to add a special "start" event to computations. We do not pursue that course here, as it would make our models inconsistent not only with [BKP84] but with most other work on state-transition models of computation.

Given this logic, one could formulate general and specification-relative conditions on ϕ_X to guarantee that a simplification preserves the falsity or truth of a path formula consisting of a single event symbol. It is surprisingly difficult, though, to find a few simple rules that are satisfactory. The problem is primarily with FP(e), which seems to require that all runs of symbols in $\Sigma - e$ are represented in the image of a path by at least a single occurence of a symbol from $\Sigma - e$. If this is not the case, then $\mathbf{G}e$ may be violated by x and satisfied by $\phi(x)$.

The problems of finding rules directly for formulas involving events can be neatly sidestepped by transforming them into formulas involving only states, and making a

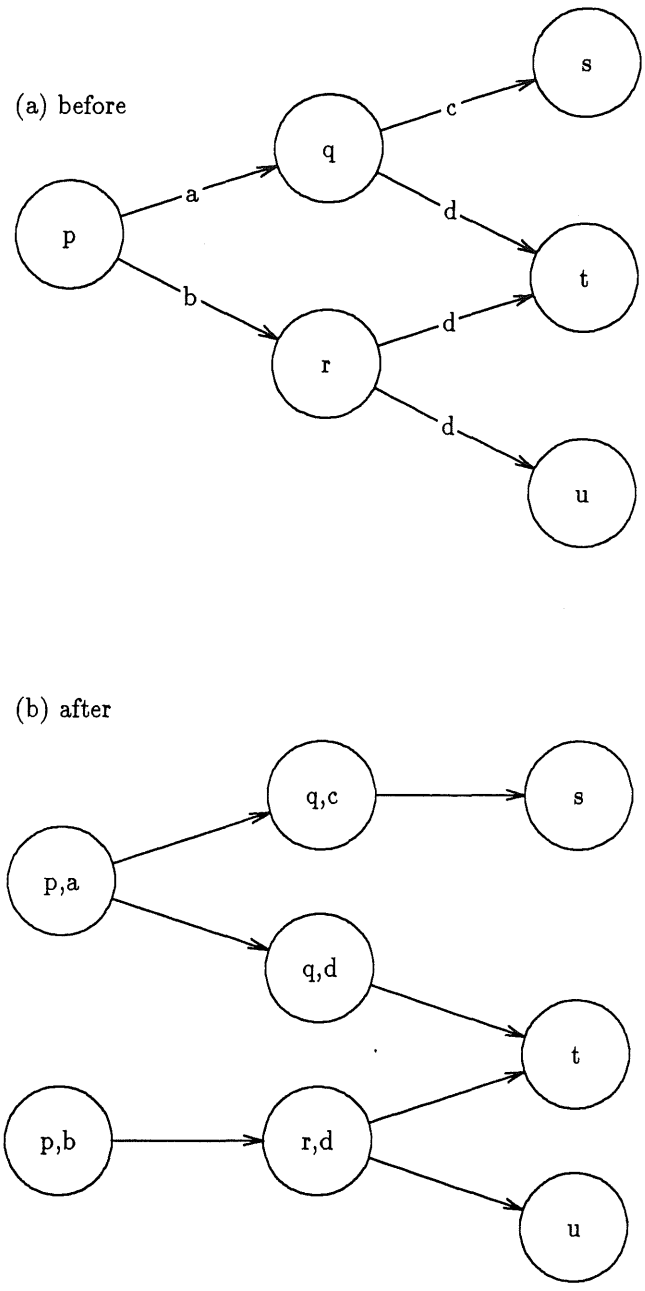


Figure 2.3: Transforming a model (a) with event symbols into an equivalent model (b) without event symbols.

corresponding transformation on the class of models (see Figure 2.3). Assume Σ is initially disjoint from the set of atomic propositions $\mathcal{P}$. We add each of the symbols from Σ to $\mathcal{P}$. Then, each state in each model is duplicated up to $|\Sigma|$ times, so that if $q \in \delta(p, \sigma)$ in the original model, $p_\sigma R q_i$ (for each copy q_i of q) in the transformed model. Moreover, $\eta(p_\sigma) = \eta(p) \cup \{\sigma\}$. It is easy to see that $x \models f$ in the transformed model iff $x \models f$ in the original model.[2]

Having transformed a model and formula with event symbols into an equivalent model and formula with only states, one can use rules from the preceding section to simplify the model. If desired, the simplified model can then be transformed back to the original form, with events. Of course one need not actually perform these transformations, provided it can be shown that the simplification of a model with states and events could in principle have been achieved in this way.

2.7 Application

Application of the rules given above is demonstrated by considering the relation between static concurrency analysis of concurrent Ada programs [Tay83b, LC89] and actual program execution. We show that static concurrency analysis is conservative with respect to the errors it is designed to detect, by showing that the (finite) reachability graph constructed by static concurrency analysis is an error-preserving abstraction of actual execution.

Chapter 4 argues that static concurrency analysis can be fruitfully combined with symbolic execution, since the state space of the former is a natural simplification of the latter. The soundness of this combination rests on static concurrency analysis being an error-preserving abstraction of symbolic execution. To make this chapter self-contained, here we consider normal program execution rather than symbolic execution. The argument is almost unchanged if *symbolic execution* is substituted for *program execution,* and *path condition and path expression* is substituted for *data values* in what follows.

Static concurrency analysis folds the state space of program execution in two respects. First, data values are ignored, so that an infinite number of possible states characterized by control point and data state are represented by a finite number of control points (flowgraph nodes). Second, regions of code that do not participate in task synchronization and communication are summarized into single flowgraph nodes. The simplification function ϕ_Q is therefore the composition of a function that

[2] A similar trick is used in [CES86] to define fairness conditions in a purely state-based temporal logic. There, the event symbols reflect the most recent process to move.

omits path condition and path expression (call it ϕ_1) and a function that merges flowgraph nodes (call it ϕ_2). A possible additional simplification is that the first-in first-out ordering of tasks in entry queues [ALR83, 9.5.15] may be ignored without risk of failing to identify potential deadlocks or race conditions; this simplification is considered separately below.

For brevity, only a single simple property is considered here. One sort of deadlock occurs when a task issues an entry call, and that entry call is never accepted by another task. We use symbol c to represent the event in which a particular task T calls entry e, and event symbol a for the called task engaging entry e. The specification formula to be considered is $c \rightarrow \Diamond a$ in the common notation of linear time logic, or $\mathbf{A}(c \rightarrow \mathbf{F}a)$ in the CTL* temporal logic (extended to include events, as described in the previous section). The property we wish to show is therefore $\mathrm{FP}(\mathbf{A}(c \rightarrow \mathbf{F}a))$.

General conditions. Omitting data values from a program execution state, leaving only a tuple of control points (flowgraph edges), is clearly a total function from states to states. (We adopt the convention that ϕ_1 and ϕ_2 are identity mappings when applied to already-simplified states.) Merging flowgraph nodes likewise results in a total function. Connectivity in the state space is preserved, since static concurrency analysis follows all paths (unexecutable as well as executable) in each flowgraph. Omitting the path condition and path expression is also loop-preserving, but flowgraph reductions may not be, so it is necessary to assume that program loops between synchronization points always terminate. A pointwise extension of ϕ_1 is an order-preserving function from paths to paths, and since the state space resulting from ϕ_1 is finite, the standard extension of ϕ_2 (Definition 7) is order-preserving.

Conditions relative to specification formulas. Decomposing $\mathrm{FP}(\mathbf{A}(c \rightarrow \mathbf{F}a))$ according to the lemmas in Section 2.5.3, we find that it is sufficient to show $\mathrm{TP}(c)$ and $\mathrm{FP}(a)$.

To facilitate reasoning about $\mathrm{FP}(a)$, we perform the transformation described in Section 2.6 above, so that c represents the state just before an entry call is performed, and a represents the state just before the entry is accepted. This matches Taylor's original description of the technique [Tay83b], in which tasking actions are represented by nodes rather than edges in a flowgraph.

Condition $\mathrm{TP}(c)$ is easy, since static concurrency analysis retains flowgraph nodes in which synchronization activities occur and sometimes uses them to represent other adjacent nodes. However, the sufficient conditions for $\mathrm{FP}(a)$ are not satisfied. The simplified state representing engagement of an entry may also represent some flowgraph nodes following the node at which the entry call becomes engaged. a must

instead be interpreted as being true of states in which the rendezvous is engaged. Alternatively, we can use the special coarsening rule above if states preceding the *accept* statement are merged.

Static concurrency analysis as described by Taylor [Tay83b] is designed to detect a fixed class of synchronization errors. This class of errors can be associated with a set of specification formulas in temporal logic, and the simplifications justified for each. This is a fairly trivial exercise, since the flowgraph reductions only summarize away those portions of the flowgraph that do not affect the synchronization structure of a program. However, the exercise does reveal some hidden assumptions, including the condition that all loops not cut by task interactions are assumed to terminate. Moreover, it shows that the procedure for detecting deadlock, as originally described, is incorrect. This procedure was to simply check the reachability graph for terminal states. If an *accept* statement has a guard condition, then it represents some states in which an entry will be accepted and some in which it won't, depending on the state of variables. When the states are merged together, an originally dead state may be mapped to a state with out-edges.

An interesting possibility to consider is using specification formulas to guide flowgraph reduction. Olender and Osterweil [OO86] have extended data flow analysis to check temporal properties expressed in a rich specification language; Morgan and Razouk [MR87] have applied a temporal logic checking procedure to Petri net reachability graphs. If static concurrency analysis were similarly extended, the user would associate propositional variables with various points in a program (not necessarily synchronization points). The rules given here could then be used to determine which flowgraph nodes could be merged with their neighbors, and the rule to use (*AND* or *OR*) in merging.

Ignoring FIFO order in entry queues. Entry queues are used in Ada to guarantee first-in first-out order in accepting multiple tasks at a single entry. Young and Taylor claim that entry queues may be represented by unordered sets without risk of hiding deadlocks. Representing ordered queues by unordered sets has the effect of merging many states, and adding some edges (corresponding to accepting a task that is not at the head of the queue). This simplification does not merge states which differ with respect to the propositional variables in the specification, and is clearly a function from states to states that preserves all of the original edges, so it is indeed an error-preserving abstraction with respect to (this simple version of) deadlock detection.

Shatz and Cheng [SC88] make a stronger claim that the order of entries in an entry queue is a "pure run-time property," i.e., that ignoring the order of entries introduces no inaccuracy in static analysis, neither hiding errors (*optimistic*) nor

causing spurious error reports to be generated (*pessimistic*). This is in contrast to the way that such claims have been relativized to specification formulas in this chapter. Indeed, it turns out that the theorem proved by Shatz and Cheng (that, in a particular state in which two or more tasks are waiting in the queue, the order in which they arrived is independent of program semantics) is not equivalent to the more general claim.[3] This is shown by considering absence of starvation conditional on a weakly fair task scheduler, i.e., one in which every task that is not blocked must eventually get a time slice. Pessimistic inaccuracy is introduced in this case, since starvation may be detected even when it would be prevented by queue ordering. To show the absence of pessimistic inaccuracy using the system described here, one would attempt to derive $\mathrm{TP}(\phi, \mathbf{A}f)$, where f is the conditional statement. Recall that there is no rule for deriving $\mathrm{TP}(\phi, \mathbf{A}f)$, regardless of the form of f — precisely because simplifications that introduce new paths, as ignoring entry queue order does, are liable to introduce pessimistic inaccuracy.

Lam and Shankar [LS84] have also given rules for simplifying state-space models. Some of their rules for preserving liveness properties correspond to our general rules for verifying temporal properties, but Lam and Shankar give no rules relative to particular specification formulas. The general rules must therefore be overly restrictive (for the uses we have in mind; they may be appropriate for verifying communication protocols). The property of faithfulness described by Lam and Shankar is too strong a condition to allow ignoring FIFO order in entry queues because it depends on a property of well-formedness that is not satisfied by simplifications that introduce unexecutable paths.

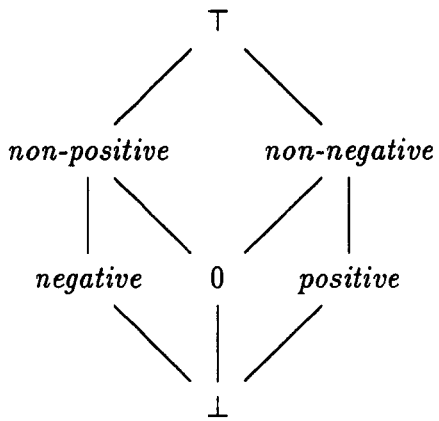
2.8 Abstract interpretation

The state-space models considered here are operational in the sense that there is an explicit notion of state and state transition. Operational models are quite intuitive, and we believe the simple notion of a function ϕ mapping states to states will facilitate reasoning about error-preserving abstractions. On the other hand, denotational semantics has some advantages over the operational approach. The analogue of error-preserving abstractions in a denotational setting is *abstract interpretation* [AH87]. We compare the two approaches here, and show how abstract interpretation can be used to define suitable abstraction functions.

[3] The claim and the theorem would be equivalent if only properties of states, and not temporal specifications, were allowed. Although Shatz and Cheng do not explicitly state this restriction, the properties they consider all fall into the class of properties for which entry queue order is indeed irrelevant.

2.8.1 Background: Abstract interpretation

Abstract interpretation is a family of techniques for giving an alternative, approximate semantics to programs. An approximate semantics is typically formed by compressing an infinite set of possible data values into a small set of representatives, organized in a lattice ordered by set inclusion.[4] For instance, the set of all possible integer values can be replaced by values from the following lattice:



In the lattice above, the bottom element $\perp$ represents an uninitialized value, or the set containing no integer values. The top element $\top$ represents all possible integer values. Intermediate elements represent positive integers, zero, negative integers, non-positive integers, and non-negative integers. Each element is approximated by the values above it in the lattice. Operations in the programming language are approximated as well, so that for instance the usual + operation might be replaced by a function that applies the usual rules, e.g., *positive* + *non-negative* = *positive* but *positive* + *negative* = $\top$.

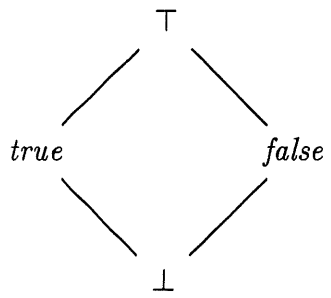
Approximate values must be included in the lattice for the same reason that the abstraction function ϕ could not be required to be onto: Folding an infinite domain of values into a finite domain, like folding an infinite state space into a finite state space, inevitably results in the introduction of representatives for values that cannot be obtained on any actual execution of the program. In our state space models we allow introduction of these impossible states but insist they not hide an error. In abstract interpretation, approximate values represent both values that may occur in actual execution and some that cannot.

[4]Other orderings are possible. For a more thorough, and very lucid introduction, see [AH87].

Abstract interpretations are devised in a manner that allows computation of fixed points in a finite number of steps. (Thus, unlike error-preserving abstraction, abstract interpretation allows one to reason explicitly about some kinds of analysis procedures.) It is intended that these fixed points be valid approximations of all possible program executions. This correctness property is established by proving a relation between the abstract semantics, including the abstract program functions like + in the example above, to the usual semantics of a language. The concrete value domain is extended to a "collecting" semantics in which each program state includes a representation of all prior states. This allows relating the two domains by showing that abstract values are supersets of the concrete values they represent.[5]

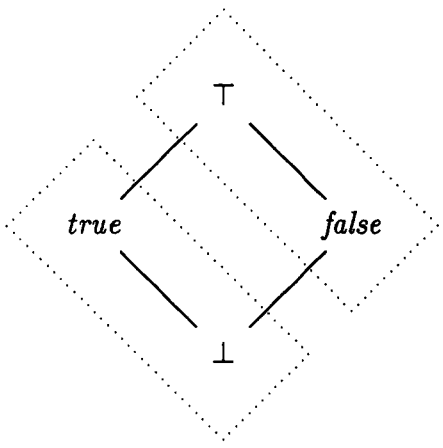
2.8.2 Interpreting propositions

Abstract interpretation is closely related to the abstraction function ϕ upon which we have been placing requirements. Although states in the state spaces considered are not necessarily organized as a lattice, it is not too difficult to adapt an abstract interpretation into a mapping from states to states. Boolean values in an abstract interpretation can be organized in the lattice:



In this lattice, the top element $\top$ represents "don't know," or collections of states in which a boolean expression may be sometimes *True* and sometimes *False*. This is analogous to the situation in which ϕ folds together states that differ in the value of an atomic proposition. Atomic propositions in state space models, as treated here, take on only the values *True* and *False*. To satisfy the basic rules concerning the value of an atomic variable in a folded state, it is merely necessary to treat "don't know" as "whatever is worse." Thus, if we are obliged to preserve the falseness of a propositional variable, we group $\top$ with *False*:

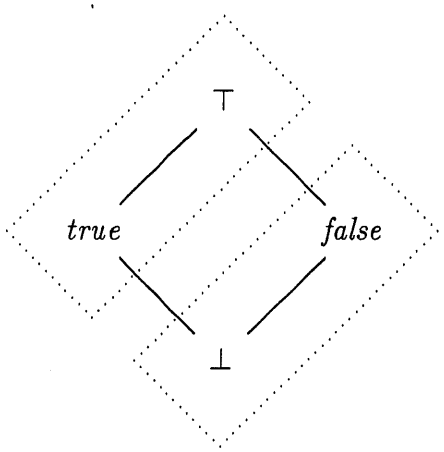
[5]More precisely, abstract values are related to concrete values by a pair of functions, α : concrete $\rightarrow$ abstract and γ : abstract $\rightarrow$ concrete. The requirements are $\alpha(\gamma(x)) = x$ and $\gamma(\alpha(x)) \sqsupseteq x$ where $\sqsupseteq$ is the lattice ordering.



In other words, the atomic proposition a associated with a binding of boolean value a to $\top$ is *False*, since the proposition *might* be false in a corresponding concrete state.

The bottom element $\bot$ may represent a value in an unreachable state, or it may represent an undefined value formed by some illegal computation (e.g., use of an uninitialized variable, or division by zero). In the former case, it does no harm to assign the value *True* to atomic propositions in unreachable states. In the remaining cases, $\bot$ represents an error outside the class of errors the current analysis technique is intended to find. (If it were in the purview of the current analysis, the specification formula should be violated rather than undefined.) If we assume that another analysis technique will be used to detect these errors, we may once again interpret $\bot$ as *True*.

If we require an abstraction function ϕ to preserve the truth of some proposition, we group $\top$ with *True* and $\bot$ with false:



The correspondence between abstract interpretation and error-preserving abstraction is very close. For specifications in a standard propositional logic, everything that can be done with error-preserving abstractions can be done as well in a denotational framework using abstract interpretation. In fact, the presence of $\top$ in value domains makes abstract interpretation somewhat more refined.

Abstract interpretation is essentially a special case of proving equivalence between two dentotational semantics[6], so it is applicable to any structure with a denotational semantics. This includes process algebras for modeling concurrent execution, which in modern expositions are given denotational semantics with value domains ordered by determinacy (see, e.g., [Hen88]). In principle, abstract interpretation might be used also to reason about simplifying models for analysis of sequencing properties. This would require not only denotational semantics for the models, but a suitable lattice of denoted values for specification subformulas.

Abstract interpretation seems to be an appropriate theoretical tool for abstracting the values of variables. It is less suitable for reasoning about control, or about models other than conventional program texts. The more operational approach of devising a mapping ϕ from states to states is well suited for those problems. In particular, when specifications are stated in temporal logic, it is easier to apply the rules given here than to reformulate the requirements using denotational semantics.

The example above, in which we justified ignoring the order of Ada entry queues, would have been greatly complicated by a denotational semantic framework, especially if the specification were a temporal logic formula. Fortunately abstract interpretation and error-preserving abstraction are easy to combine. One can use the

[6] According to Mycroft [Myc87, page 205].

theory of abstract interpretation to show the safety of inferring boolean values, and use the rules given here to show that errors are preserved when those boolean values are combined in temporal formulas. Perhaps the lemmas proved in this chapter, or close analogues, can be restated and proved in a denotational framework; this is left for future work.

2.9 Summary

Perfect accuracy in a software fault detection technique is possible only with infinite effort. All practical techniques therefore compromise accuracy in order to control cost. An important class of techniques is those that allow inaccuracy only in the pessimistic direction, i.e., those that may reject a correct program but which will never allow an error to escape detection. Program proof belongs to this class, as do a variety of analysis techniques that mechanically "prove" the absence of certain classes of errors.

In devising or modifying a fault detection technique, one often needs to justify a claim of the form, "The following simplification will not cause any errors to go undetected." This chapter has introduced a formal framework for making and defending such claims when the simplification consists of "leaving out details" in a state-space model of execution. Sufficient (but not necessary) conditions for showing that a simplification preserves errors are given with respect to specification formulas in propositional logic and a large subset of propositional temporal logic.

The claim must always be relative to a specification, since all but trivial simplifications are bound to change *some* properties of a model. This is not usually a problem for techniques designed to detect a fixed class of errors (e.g., deadlocks or data flow anomalies). For techniques that allow checking programmable specifications, the rules given here can be used to guide simplification of the model before analysis. The property of one model being an error-preserving abstraction of another is particularly useful when multiple models (or levels of detail) are to be combined in a hybrid analysis technique, as described in the Chapter 4.

Chapter 3
Parceling the analysis of concurrent systems

3.1 Introduction

This chapter and the next consider the problem of detecting faults in the synchronization structure of concurrent programs. Such faults include, for instance, deadlocks and race conditions. As with fault detection in general, there is no possibility of devising a perfect technique for recognizing synchronization errors. In its most general form the problem is undecidable, and it remains intractable even for severely restricted problems. Practical approaches to analyzing concurrent software require a delicate balance to make the problem tractable for typical cases without introducing so much inaccuracy that the analysis becomes worthless. We consider refinements to a particular technique, static concurrency analysis. This chapter examines the problem of controlling combinatorial explosion in static concurrency analysis, at the cost of some additional inaccuracy. The primary focus is on dividing a large analysis problem into smaller problems, incurring some pessimistic inaccuracy. Sampling a folded model, thus incurring optimistic inaccuracy, is also considered. Chapter 4 addresses the complementary problem of refining the results of static concurrency analysis, reducing pessimistic inaccuracy at the cost of some additional effort.

3.2 Background

3.2.1 The problem

Analyzing concurrent software is difficult. Complexity is inherent in the problem, rather than being a flaw in a particular model or analysis technique. Taking exposure to deadlock as a representative problem for analysis, it is easy to see that it is undecidable in the general case. One may reduce the halting problem to a deadlock

problem as follows: Create a system with three tasks (processes). Two of these are as follows (in Ada):

```
task body T1 is
begin
 loop
   T2.E1;
   T2.E2;
 end loop;
end T1;

task body T2 is
begin                                                    10
 loop
   accept E1;
   accept E2;
 end loop;
end T2;
```

A third task is added in a way that will expose the system to deadlock if and only if a program P halts. Assuming P can be called as a procedure (so "halts" means "eventually returns from a procedure call"), we add a third task as follows:

```
task body T3 is
begin
 P;
 T2.e1;
end T3;
```

Practical analysis techniques, based typically on finite-state models of programs, accept some (pessimistic) inaccuracy to make the problem decidable. However, even the finite-state problem is intractable for large systems. NP-hardness or Pspace-completeness results have been shown for analysis of finite state versions of CSP [Lad79], Petri net reachability analysis [Pet81, Val88], and synchronous communicating finite state machines [Smo84], as well as severely restricted Ada programs [Tay83a]. Thus, no perfect remedy can exist,[1] where a perfect remedy would be one that neither limited the domain of applicability of the technique, nor introduced (more) spurious error reports, nor failed to report a possible error. Every remedy must compromise in one or more of these ways.

[1] Excepting an affirmative answer to the $P = NP$? question.

3.2.2 Static concurrency analysis

Static concurrency analysis [Tay83b, Apt83] can be used to detect synchronization errors such as deadlock and parallel update of shared variables in concurrent programs where the model of synchronization is some form of rendezvous, as in Ada [ALR83], CSP [Hoa78], or Hal/S [Mar77]. Taylor [Tay83b] has described a concurrency analysis algorithm in a form suitable for Ada; Apt [Apt83] has applied essentially the same technique to CSP. It should be equally applicable to design notations utilizing rendezvous for synchronization. Language-dependent examples in this dissertation are based on Ada.

Static concurrency analysis builds a rooted directed graph of *concurrency states.* A concurrency state summarizes the control state of each of the concurrent tasks at some point in an execution, including synchronization information, while omitting other information such as data values. Directed edges in the concurrency state graph indicate which states may follow each other in executions of a program. A path from the root node to any node in the graph is called a *concurrency history,* since it captures a sequence of synchronization events that may occur in a program execution.

Some tasking-related errors are manifested in a concurrency state, while others are properties of multiple states in a concurrency history. An infinite wait, for instance, is manifested as a concurrency history ending in a state (therefore a leaf node) with one or more tasks still active. Concurrent update of a shared variable shows up as a state in which more than one task is capable of writing to that variable before the next synchronization event.

The primary strength of static concurrency analysis is that it examines all possible synchronization patterns. Errors cannot be masked by differences between the testing environment and production environment, as they might be in dynamic analysis. This property is especially crucial in validating concurrent software, since synchronization patterns are partially determined by a scheduler, and may be sensitive to timing.

Static concurrency analysis represents one possible compromise between computational effort and accuracy. In terms of the framework discussed in Chapter 1, static concurrency analysis *folds* the infinite state space of a concurrent program into a finite state space, incurring as a consequence some pessimistic inaccuracy. The folding is defined only for programs in which the number of tasks is bounded. Analysis of the folded model is no longer undecidable, but it remains expensive enough (exponential in space and time) that only small programs can be analyzed in whole.

3.2.3 Concurrency analysis algorithm

This section describes Taylor's algorithm [Tay83b] for static concurrency analysis in simplified form. A concurrent Ada program is represented by a set of flowgraphs, one for each task. In Ada, the top-level procedure (main program) is also considered a task. These flowgraphs are summarized into a set of finite state machines with synchronous communication. The set of all possible execution sequences of these state machines is represented by the concurrency graph, which is also a finite state machine. This model of individual process graphs and their product graph is equivalent to networks of finite state processes as considered by Clarke and Grümberg [CG87]. Many similar models appear in the literature, including the regular subset of Milner's CCS [Mil80] and theoretical CSP [BHR84].[2]

To simplify exposition of the relation between an Ada program and its representation as a set of state machines, we impose several restrictions and simplifying assumptions. First, we assume that all tasks begin execution simultaneously and terminate simultaneously, if at all, so that the only synchronization that must be modeled is the Ada rendezvous. This will also allow us to omit the main procedure from our models unless it participates in rendezvous. (Modeling initiation and termination of tasks adds to the complexity of analysis but does not change it in any essential respect.) Further, we assume that procedures are not recursive, so that procedure calls can be modeled by copying a procedure body into a task at the point of call. (In practice recursion causes problems only if recursive procedures declare tasks or perform inter-task communications, in which case it can be modeled with some restrictions; see [Tay83b]). Dynamically identified tasks and communications (arrays of tasks, pointers to tasks, and entry families) are prohibited. Finally, we ignore guards on *accept* statements. In principle, a program with guards can be transformed into an equivalent program without, although in practice this would create unnecessarily complex models.

Under the restrictions imposed above, it is easy to see that each task flowgraph is essentially a finite state machine, or labeled transition system. More precisely, the dual of each flowgraph is a finite state machine, since flowgraph nodes represent actions and flowgraph edges represent states. (This is why Floyd-style assertions appear on flowgraph edges and not nodes.) Each task can progress independently through its flowgraph until it reaches a synchronization point. It will be simpler in what follows to consider the state machine representation (nodes = states) rather than the flowgraph representation (nodes = actions).

[2] "Theoretical CSP" is the term commonly used for the algebraic models of communicating sequential processes investigated by Brookes, Hoare, and Roscoe, to distinguish them from the earlier work by Hoare which was also called CSP.

The state machines must be modified in two ways before analysis. First, a single entry call or accept in Ada must be split into multiple actions, representing steps in a rendezvous. An entry call is represented by an action to *engage* the rendezvous, followed by an action to *finish* the rendezvous. During the time a rendezvous is engaged, the calling task is blocked, so the two actions representing the calling side of a rendezvous are separated by a single node with no other edges. Complementary actions for engaging and finishing a rendezvous are used to represent a single *accept* statement, but in this case intervening nodes may represent tasking or non-tasking actions, since a block may be nested within an Ada *accept* statement.

We will use the following notation in diagrams:

A is the beginning of a rendezvous at entry A, offered by the calling task.

$\overline{A}$ is beginning of a rendezvous (engagement) at entry A, offered by the accepting task. A and $\overline{A}$ are complementary actions. They occur jointly or not at all.

A_E is the end of a rendezvous at entry A, offered by the calling task. This is the only action a calling task can take after engaging a rendezvous.

$\overline{A}_E$ is the end of the rendezvous at entry A, offered by the accepting task. A_E and $\overline{A}_E$ are complementary actions. A calling task blocks waiting for A_E, but the accepting task is never blocked at $\overline{A}_E$.

Once task communications have been broken into their atomic steps, one could in principle construct a product state machine directly from the representations of individual tasks. However, it is much better to first reduce the individual state machines as far as possible. Taylor [Tay83b] gives one set of rules for reducing the individual state machines, and Long and Clarke [LC89] give a modified set of rules which usually result in smaller models. Long's models are called "task interaction graphs," to emphasize the fact that all edges in the summarized model represent interactions between tasks, and all non-tasking actions are summarized within nodes.[3]

Figure 3.1 illustrates how a program flowgraph is summarized. A single node in the summarized representation takes the place of a set of flowgraph paths up to a set of possible synchronization actions. The representation used here is essentially that of [LC89], except that to keep the examples small we omit initiation and termination and we allow nodes to have in-edges with different actions. (If the rules of [LC89] were followed precisely, the final node in diagram 3.1 would be split into three, to account for possibly different non-tasking activities following the task interactions on each of the two program branches and preceding termination.)

[3]Some readers may notice that the distinction between internal nondeterminism (conditional branches in a single task) and external nondeterminism (Ada *select* statements) has been lost. This distinction is essential for recognizing deadlock. Fear not: The problem has been glossed over for the moment, but it will be cleared up in Section 3.3.3 below.

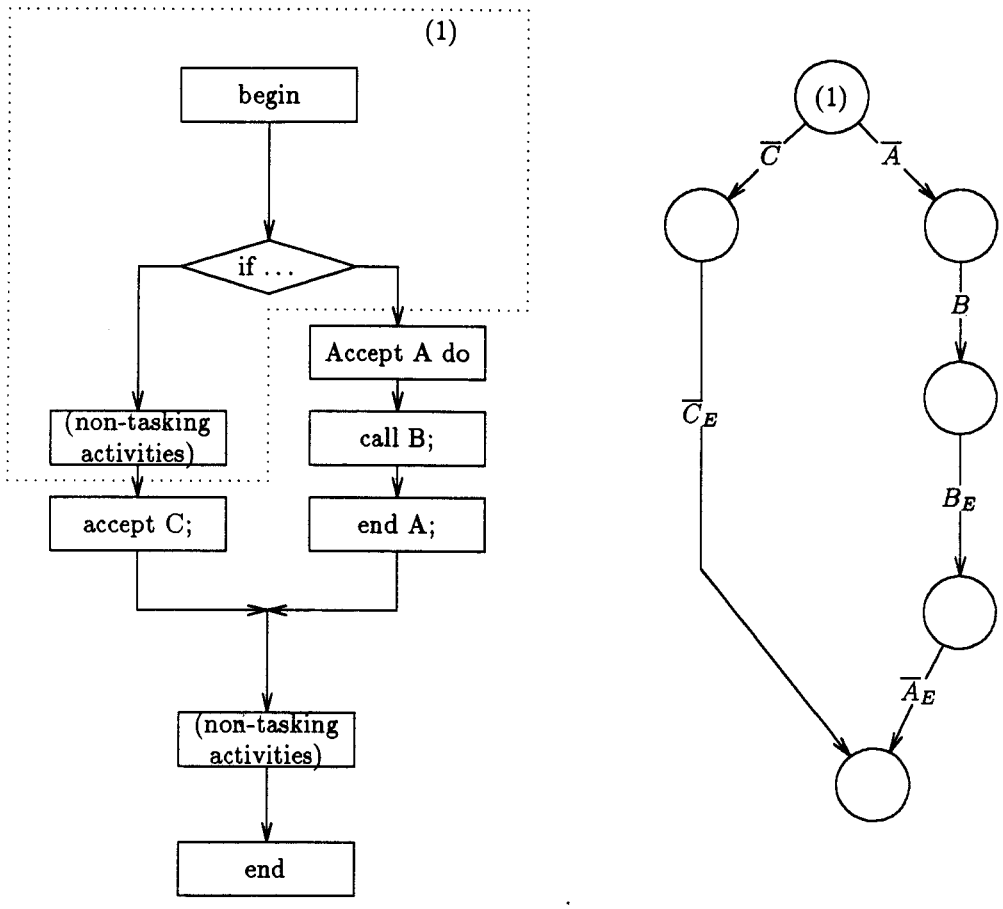


Figure 3.1: A program control flow graph and its reduced finite state machine (task interaction graph) representation. All of the region enclosed in the dotted line is represented by the single node (1). Symbols on edges in the reduced representation represent tasking actions. $\overline{A}$ is the beginning of a rendezvous on entry A, from the client side, and is complementary to a A action from a calling task. $\overline{A}_E$ is the action an accepting task takes to finish a rendezvous, and is complementary to a A_E action by the calling task.

The modeling procedure can be intuitively viewed as moving tokens through the task interaction graphs. Analysis begins by placing a token at the initial node of each task. A concurrency state is a list of the current positions of all tokens. New states are generated by advancing a pair[4] of tokens together at synchronization points. As in similar models, we perform only one joint action even when multiple, independent joint actions are possible, on the grounds that an observer that can make only a single test at each moment cannot distinguish between interleaving and true simultaneity.

It is easy to see that the number of states in the concurrency graph is bounded by the product of the number of nodes in the reduced flowgraph representation of each task. For a system of T tasks, each with n nodes, the concurrency graph will have at most n^T states. One might hope that behavior in typical cases would not approach this upper bound, but experience with static concurrency analysis [Wam85] and similar techniques suggests that the exponential upper bound is an accurate predictor of actual behavior. Static concurrency analysis can be applied only to small systems in whole.

3.3 Parceling

Since a single, global analysis of a program with many tasks is hopeless, we must consider ways of parceling the global analysis problem into a number of sufficiently small local analysis problems, and then combine the local results. The ultimate parceling would be individual analysis of each task, such as the Lamport-Owicki-Gries approach to separating local process proofs from a cooperation proof [OG76, Lam83] or the compositional temporal logic proof system described by Barringer, Kuiper, and Pnueli [BKP84]. Applying these proof systems requires considerable human ingenuity (or else suffers from even worse combinatorial explosion than explicit modeling). For an automatable technique like static concurrency analysis, it is not necessary to achieve completely independent analysis of each task. It is enough to divide a large system into parcels of a few tasks each, provided each parcel is small enough for practical analysis.

[4]We need move only pairs because rendezvous is the only task interaction considered, as per the simplifying assumptions listed above. Proper modeling of initiation and termination of tasks requires moving single tasks and/or larger groups of tasks in some situations.

an intractable analysis problem remains. We therefore propose a parceling method with an increased domain of applicability as compared to the biconnected components method. We pay for increased applicability in two ways: The programmer will be required to guide the parceling, and additional spurious error reports may be produced.

A *weak monitor* is a group of procedures, tasks, and packages identified by the programmer as a module and alleged to have certain properties (described below) which allow it to be analyzed in isolation. During the analysis of one weak monitor, the required properties are assumed for other alleged weak monitors, and verified for the currently considered weak monitor. An acyclic dependency relation among weak monitors makes the entire analysis procedure sound.

A subset of the task entries comprising a weak monitor may be called by code outside the monitor; we call these the "entries" of the monitor (by analogy to task entries). For an entry which is a task entry, the notions of "called", "engaged", and "finished" are used exactly as in tasking. For procedures, we treat a procedure call as if it were a task entry call immediately engaged, and as if the return from a procedure were the end of a rendezvous. (Reentrancy of procedures requires a modified analysis procedure, described below.) A weak monitor must have the following properties:

(A) No data item accessed or updated by any task or procedure in the monitor is accessed or updated by any procedure or task outside the monitor.

(B) From any program state in which no entry of a monitor is engaged, but some entry has been called, there must follow eventually a state in which either the call is retracted (in the case of a conditional entry call) or else the call is engaged.

(C) From any program state in which N calls are engaged, $N \geq 1$, there must follow eventually a finish of an entry call.

The reason for the name, *weak monitor,* can now be explained. The first property described above is part of the conventional purpose of a monitor, i.e., to assure mutual exclusion to data. The second and third properties above are satisfied by true monitors, which strictly serialize calls, but the weak monitor properties are weaker in that they allow (but do not promise) multiple simultaneous calls, provided *termination of a rendezvous may never depend upon other client activities during the rendezvous.* (Starvation, in which one task waits forever while other tasks are served, is not precluded.) This is a strong enough condition to allow us to analyze weak monitors separately from their clients.

The properties of a weak monitor make it suitable as an implementation of a server in a client-server system architecture. From the point of view of a client, any concurrency within the server is invisible. A server implemented as a weak monitor appears to the environment as a set of procedure calls.

One more property must be specified before we consider analysis. We say weak monitor M_1 *uses* weak monitor M_2 if M_1 calls any entries of M_2. The relation $uses(X, Y)$ must form a DAG. This does not require the *uses* graph of an entire program to be a DAG, but any cycles in the (directed) *uses* graph of a program must be confined to a single weak monitor.

3.3.3 Internal and external nondeterminism

Before considering the analysis procedure for weak monitors, we must fill in a detail that was glossed over in the description of the models used in static concurrency analysis. Two varieties of choice must be modeled, corresponding to what is sometimes called *internal* and *external* nondeterminism [Hen88]. One kind of choice results from internal program decisions (conditional branches) which cannot be determined from the information present in the model (e.g., they may depend on input data). This results in internal nondeterminism in the model, since the choice is considered to be arbitrary and independent of the states of other tasks. The other kind of choice occurs in communication, when a single task is willing to communicate with any one of several other tasks (e.g., using a selective *accept* statement). This is *external* nondeterminism, since it depends on the state of other tasks.

It is necessary to distinguish between the external nondeterminism of

```
select
  accept E1;
or
  accept E2;
end select;
```

and the internal nondeterminism of

```
if c then
  accept E1;
else
  accept E2;
end if;
```

The first program fragment exhibits external nondeterministic choice. If the environment provides action $E1$ but not action $E2$, then $E1$ will be taken. The second

represents internal nondeterministic choice. If the environment provides action $E1$ but not action $E2$, the task may deadlock because it has chosen to accept only $E2$.

There are two common strategies for representing the distinction between internal and external nondeterminism. One approach is to represent internal nondeterminism by extra edges in the state machine model, either by explicitly representing some internal actions, or by duplicating some edges representing task interactions in order to represent the set of states after the next sequence of internal moves. This approach is used in the Milner's Calculus of Communicating Systems [Mil80]. The alternative is to use an additional property of states, such as *refusal sets* [BHR84] or *acceptance sets* [Hen88].

Edge groups in Long and Clarke's Task Interaction Graph model are a representation of acceptance sets. They have no effect on the structure of the concurrency graph, but are essential to recognizing deadlock. A potentially dead state may be represented by a concurrency graph node with out-edges. Potential deadness is recognizable from a choice of acceptance sets, one from each task, such that the union of all does not contain any pair of complementary actions. The analysis procedure described below uses acceptance sets, and the prototype system described in Appendix A uses task interaction graphs.

3.3.4 Analysis procedure

The properties of a weak monitor are such that a client may be analyzed as if calls on weak monitors were calls on procedures which do not perform tasking activity. The properties guaranteed by a weak monitor (and verified by the analysis) provide that each call can eventually return, whether or not other clients or tasks within a client can progress in the meantime.

Two sets of modifications must be made to the standard concurrency analysis procedure. First, we consider how a weak monitor can be analyzed separately from its clients. Then, we consider how a client can be analyzed separately from the weak monitors it calls upon. These two sets of modifications can be applied together when a weak monitor is a client of other weak monitors.

The promises made by a weak monitor to its clients are designed to be independent of the particular pattern of calls by a client (or multiple clients). We wish to simulate a completely unpredictable set of clients. In particular, we don't know how many clients will make calls on a weak monitor or how their actions may be coordinated. Assume that the *count* attribute[5] is used nowhere within the weak

[5]In Ada, the *count* attribute is used to determine how many tasks are waiting on an entry queue. It could be accommodated in the analysis by allowing it to take an undetermined value, but of

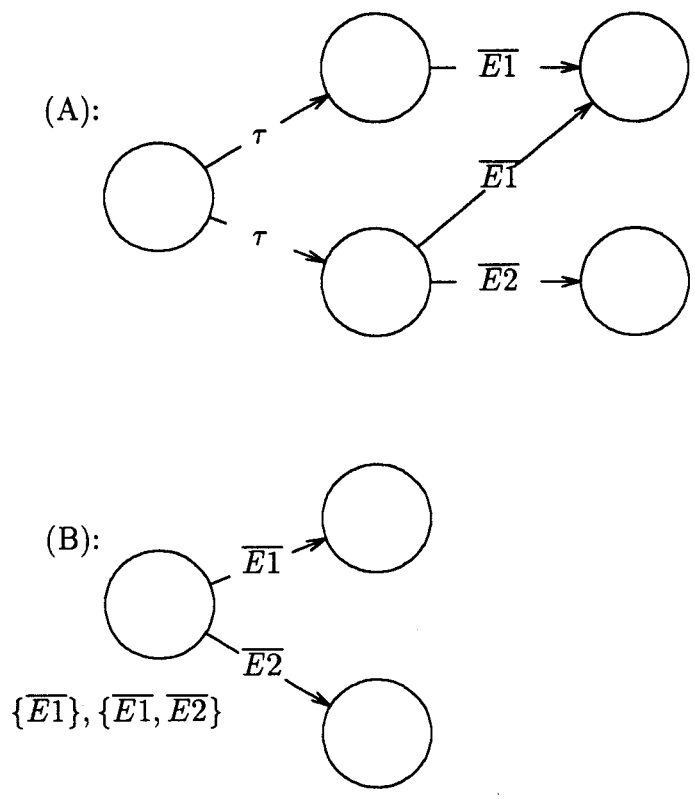


Figure 3.2: Representations of internal nondeterministic choice. The upper machine uses a null action τ to enter a state which can only accept $E1$ or a state which can accept either $E1$ or $E2$. The lower machine omits the internal actions leading to the choice, but explicitly represents the sets of actions acceptable in each case. The representation in the lower machine is called *acceptance sets*[Hen88], and is equivalent to edge groups in task interaction graphs.

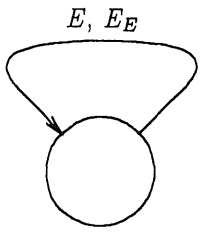


Figure 3.3: Representation for N clients of a weak monitor. The acceptance sets represent the fact that a client need not be willing to engage an entry, but it can never refuse to finish a rendezvous.

monitor. Then, for analysis purposes, *a state in which $N > 1$ calls are waiting on a particular task entry is indistinguishable from a state in which 1 call is waiting on that entry.* This observation allows us to simulate an arbitrary number of clients by a single pseudo-task. The pseudo-task, unlike a real task, may *engage* one entry call after another without an intervening *finish* of an entry call.

A one-state machine to represent an arbitrary number of clients calling a single entry of a weak monitor is illustrated by Figure 3.3. This is a trivial folding which completely ignores the internal states of clients. It allows the client to engage and finish entry calls in any order whatsoever. This clearly has the desired effect with respect to reaching all potentially reachable states of the weak monitor. That is, the folding ϕ which reduces a set of clients to a single state is total and preserves connections in the concurrency graph, in line with the general requirements of Chapter 2 for error-preserving abstractions.

Whereas the number of clients waiting on a task entry can be ignored, beyond distinguishing *none* and *at least one,* any number of clients may simultaneously engage a reentrant procedure in the interface of a weak monitor. This may be modeled as follows: A counter is associated with each flowgraph node in the procedure. Each counter may take the values $0, 1$, and ω, where ω represents two or more instantiations at the same control point. Concurrency states will include the values of all counters, so this procedure will be practical only if procedures in the interfaces of weak monitors are simple. Progress in the procedure is modeled by decrementing one counter and incrementing a counter in the next node. Incrementing ω results

course this could introduce additional spurious error reports. Its most common use is to distinguish between no pending calls and one or more. This use could be accommodated at the cost of some added complexity in the model. Fortunately, use of the *count* attribute appears to be rare in practice.

in ω, and decrementing ω results in *both* 1 and ω (i.e., two different states in the concurrency graph).

When the counter in node is 0, its only acceptance set is {} (it cannot engage in any action). When the counter is 1 or ω, the normal acceptance sets of the reduced representation of the procedure are used. When a procedure interface is used to impose sequencing on calls to a weak monitor, spurious reports of potential deadlock are likely.

3.3.5 Analyzing the concurrency graph.

Besides checking for deadlocks and variable serialization errors, one must analyze the concurrency graph to verify the two sequencing conditions of a weak monitor. These conditions must be verified so that they can safely be assumed when analyzing other weak monitors.

Potential deadlocks can be detected by using the using the acceptance set (edge group) representation of each task, including the representation of clients. If a deadlock occurs when the server is not attempting any interaction with the client, this will be detected as usual. If the server is attempting to finish a rendezvous, deadlock is not possible. Thus, the only thing we must ensure is detection of deadlocks when the server is attempting to engage a rendezvous. This could happen, for instance, if clients are only interested in engaging entry $E1$ but the server is only willing to engage entry $E2$. It could also occur if all clients are engaged in a rendezvous, but the server is unwilling to finish a rendezvous until another is engaged.

If the server is attempting to engage a rendezvous, then a state can be dead only if no client task is willing to engage a rendezvous at the same entry. Refusal to engage a rendezvous is represented by an acceptance set that does not include A, for some entry A. The one-state representation of clients in the folded model always has an acceptance set that does not include A. Thus, the image of a dead state under ϕ is clearly a dead state. (The image of many non-dead states may also be dead, but only if the weak monitor makes improper assumptions about the behavior of its clients.)

Since dead states are recognizable in the folded model, we can assume absence of deadlock when performing the other analyses and assume all other errors occur on infinite executions. Adopting the temporal logic of Chapter 2, we associate a propositional variable a with the event of engaging an entry. It is possible (and efficient, using the algorithms of [CES86]) to mechanically verify the restricted branching-time temporal logic predicate $\mathbf{AGAF}a$. (If there are several entries, the analysis is performed separately for each.) This predicate states that every infinite execution

involves engaging the entry an infinite number of times. We claim that, in the absence of deadlock, this property is enough to establish the other properties of a weak monitor. Nor is it too strong a property to require of a weak monitor: If false, the server may spin internally without ever providing service.

Property B: From any program state in which no entry of a monitor is engaged, but some entry has been called, there must follow eventually a state in which either the call is retracted (in the case of a conditional or timed entry call) or else the call is engaged.

In the absence of deadlock, property (B) follows directly from infinitely repeated service.

Property C: From any program state in which N calls are engaged, $N \geq 1$, there must follow eventually a finish of an entry call.

We observe that, provided there is a finite number of tasks in the client or server, a rendezvous can be engaged at a particular entry of the weak monitor an infinite number of times only if it is finished an infinite number of times, and vice versa. Thus property (C) also follows from the temporal property above.

3.3.6 Analyzing clients

A client of a weak monitor is analyzed as if calls on the weak monitor were calls on procedures which do not perform tasking activity. Properties B and C above provide that each call will eventually return, whether or not other clients, or tasks within a client, can progress in the meantime. Since these properties are verified in the analysis of a weak monitor, they can be assumed in the analysis of its clients. On the other hand, properties of weak monitors do not rule out sequences in which one client may wait forever while other clients are served.

Spurious states and paths in the concurrency graph may be generated, since details of the possible interactions with a weak monitor have been abstracted away. For instance, a true monitor could guarantee that only one client is ever engaged, and clients are served in first-come, first-serve order. This additional information is lost in the analysis, and the concurrency graph of a client may include paths which violate this constraint.

Extra states and paths in the concurrency graph may be reflected in spurious error reports, but they will not cause reports of illegal parallel actions to be missed. The prohibition against sharing variables between weak monitors is intended to ensure

that the only illegal parallel actions to be considered are those by tasks within a single weak monitor. The simulation described above is such that every possible interleaving of task executions within the weak monitor (and perhaps some impossible interleavings as well) is represented.

Implications for concurrent program design. Effectively parceling the analysis of concurrent programs depends upon the structure of the program to be analyzed. Both of the methods considered here depend on program organizations which localize the synchronization interdependencies of tasks. It may be argued that these considerations are a natural basis for concurrent program design methodologies. In fact, the localization of dependencies required for efficient analysis is a species of the well-known criterion of *loose coupling* between modules. This interplay between design and analysis of concurrent programs is exactly analogous to the interplay between program organization disciplines and the development of validation and verification methods for sequential programs. It is too early to say whether exactly the requirements of these parceling schemes will correspond to a suitable notion of "module" for concurrent programs, as concurrent programming with Ada-like rendezvous constructs is not yet widespread, but it is at least plausible that a suitable design discipline and a corresponding method for parceling analysis will develop together.

3.4 Heuristic search

The procedure for parceling analysis according to biconnected components neither introduces spurious error reports nor fails to report any errors, but is applicable to a restricted class of concurrent programs. The procedure for parceling analysis into weak monitors is applicable to a larger, but still restricted, class of programs; the price of the larger domain of applicability is the possibility of extra, spurious error reports. A third dimension of possible compromise to control combinatorial explosion is to forgo assurance that all possible errors have been detected. Although complete assurance is preferable, in practice one often resorts to techniques (including all common forms of testing) which lack this assurance.

Common testing and modeling techniques have the property that only a portion of the complete state space is actually explored. Holzmann [Hol87] has shown that partial exploration of a state-space may also be useful, at least in the design stage, for reachability analysis of protocols. This section considers how one might generate and analyze only a portion of the concurrency state graph. Random walks through the space are likely to be fruitless, but substantial benefit may still be gained by exploring portions of the state space in a systematic manner. When the objective

of analysis is to bring design errors to the programmer's attention, one reasonable approach is to concentrate effort on paths most likely to contain errors. Of course, if a perfect oracle could guide the search, we would throw away the analysis technique and keep the oracle. The best we can hope for, then, is to find a reasonable estimator of the distance (number of state transitions) between a given node and some node representing an error. Such an estimator is called a *heuristic function.*

Pearl [Pea84] describes how heuristic functions may be derived by relaxing constraints on the state generation function. In static concurrency analysis, the advancement of tokens through reduced flowgraphs is constrained by the semantics of the language being modeled. Distance between states is difficult to estimate because each movement of a token may restrict movement of other tokens in later steps. But consider a relaxed form of concurrency analysis in which tokens are allowed to move independently, unconstrained by synchronization with other tasks. In this case the distance to a certain configuration of tokens is just the sum of the distances of each token to its position in that configuration. Moreover, if interesting configurations can be described in advance, then most of the distance computation can be moved to a pre-processing stage.

This approach is demonstrated by creating a heuristic function for a particular class of synchronization errors. Suppose several tasks share a global variable v. The problem is to discover whether two or more tasks can ever write to variable v simultaneously. The distance from some concurrency state to a write-write conflict on variable v is then the sum of the two minimum distances of individual tokens to writes of v. Read-write conflicts and variable serialization errors (Task_1 reads v, then Task_2 reads v, then Task_1 writes v, then Task_2 writes v) can be detected analogously.

The distance from each flowgraph node to a node in which variable v may be written can be determined before search begins, in time linear in the number of flowgraph nodes. If no descendant of a node can write to v, the distance will be considered infinitely large. The cost of this preprocessing is insignificant compared to any computation that must be carried out for each concurrency state.

The heuristic distance estimates would be used to determine the order in which paths are explored in static concurrency analysis. If h is calculated after each state transition (for instance, if the A^* search algorithm [Nil80] is used to guide all generation of concurrency states), the individual distances can be kept in a priority queue implemented as a heap, at a cost of $O(\log T)$ operations to update the heap at each state transition, and a cost of T storage locations in each state on the search frontier. Recomputing the heuristic function h is then performed in constant time, since the two minimum-valued elements of a heap can be obtained in three accesses. A priority queue is overkill if the number of tasks is small (but in that case exhaustive analysis may be possible).

If several heuristics are used together to search for any of several different errors, they may be combined by taking a minimum of the individual h values for a node, possibly scaled to reflect their importance or accuracy.

3.5 Summary

Analysis of concurrent programs is an inherently difficult problem. Combinatorial explosion must be held in check if it is to be practical for large programs. Complexity of static concurrency analysis can be radically reduced by independently analyzing small portions of the program, provided it is structured in a suitable manner. The requirements for parceling appear to coincide with a natural discipline for modularizing concurrent programs (client server architecture), but this will not be certain until concurrent programming in Ada-like languages becomes more widespread. When parceling fails and optimistic inaccuracy is acceptable, one may search the state space rather than exhaustively enumerating states.

Chapter 4
A hybrid technique: Combining static concurrency analysis with symbolic execution

4.1 Introduction

This chapter continues consideration of static concurrency analysis. Whereas Chapter 3 considered ways of trading some inaccuracy to achieve better performance, here we consider a way of improving the accuracy of the technique at some cost in performance. Though seemingly contrary, the two directions are actually complementary.

Suppose one has a spectrum of fault detection techniques, ranging from techniques that are very cheap but subject to severe pessimistic inaccuracy, to techniques that are very accurate but very expensive to apply. Under the right conditions, it may be possible to combine the techniques as follows: A relatively cheap technique is first applied, and (possibly due to pessimistic inaccuracy) it produces a list of possible errors. Then a more expensive technique is applied, but instead of being applied in an exhaustive fashion, it is used to filter the set of possible errors. The filtered list is then filtered further by a more accurate technique, and so on. The basic idea is that, by using the more accurate techniques only to filter candidate errors generated by less accurate techniques, one may derive the benefits of the more expensive technique without the full expense of applying it in isolation.

Integrating static concurrency analysis with symbolic execution can sharpen the results of static concurrency analysis without incurring the full cost of symbolic execution. Symbolic execution can be used to filter candidate errors detected by static concurrency analysis, without danger of failing to detect errors, because static concurrency analysis is an error-preserving abstraction of symbolic execution with respect to the classes of errors it is designed to detect.

4.2 Background: Symbolic execution

Symbolic execution is a well-known technique for analyzing sequential programs. Symbolic execution preserves information about data values that is lost in static concurrency analysis. This information can be used to detect infeasible execution paths, as well as to generate test data [Cla76] and to aid in program verification [HK76, CR84, KE85]. Natural extension of sequential symbolic execution techniques to concurrent programs in a language like Ada requires simulation of a run-time scheduler to explore different interleavings of events [KG85].[1] The number of possible interleavings may be very large. The symbolic execution method outlined below is intended to accommodate a simple mapping from paths explored in symbolic execution to paths in the graph produced by static concurrency analysis, such that only interleavings that correspond to different concurrency histories need be explored.

Symbolic execution algorithm. A concurrent program is represented once again as a set of flowgraphs, with the extensions used for static concurrency analysis, but without the simplifications. (Simplifying the flowgraph for symbolic execution is discussed in Section 4.5.1.) Tokens are used to represent threads of control, and new states are generated by advancing tokens as before. Two additional pieces of information are associated with each state: A path expression and a path condition. The path expression associates symbolic values with program variables. The path condition is a boolean expression involving relations between input variables, and expresses the conditions necessary for reaching that state.[2]

Advancement of a token through a conditional branch node adds a term to the path condition, corresponding to the branch chosen. Advancement through other nodes may modify the path expression by binding new symbolic values to variables. Any state with a path condition that can be simplified to *false* can be discarded, because no input data can ever meet the conditions necessary to reach that state.

In this form of symbolic execution, as in concurrency analysis, only single tokens will be advanced, or pairs of tokens at synchronization points. Although a multiprocessor implementation of Ada might perform more activities in parallel, the results of this parallelism (or finer-grained interleaving on a uniprocessor) cannot affect the

[1] Dillon [Dil87a, Dil87b] describes an approach to symbolic evaluation which instead uses a Hoare-style proof system to verify individual tasks separately from an overall proof of cooperation. This method avoids combinatorial explosion in generating interleavings, but requires more guidance from the programmer in the form of auxiliary variables, assertions in tasks, and a global invariant assertion.

[2] The FIFO semantics of Ada rendezvous also require keeping track of an entry queue for each task; however, omission of this information in both static concurrency analysis and symbolic execution does little harm. For a discussion of this simplification, see Chapter 2, page 54.

results except in the case of conflicting access to a shared variable. Conflicting access to shared variables is detected as a synchronization anomaly by static concurrency analysis, so it can safely be ignored in symbolic execution.

In [HK76], the paths explored by symbolic execution of a sequential program are represented by a computation tree in which nodes are states and edges are program actions. For a concurrent program, the tree becomes a directed graph, since the same state may be reached through different interleavings of task executions. Computation trees in [HK76] have the property that every node has a unique path condition, induced by the unique path from the root to that node. This property is a consequence of conjoining complementary predicates to the path conditions of nodes following a conditional branch. Scheduling decisions in concurrent programs are generally nondeterministic (or modeled as such), which allows different paths to arrive at the same node. It is this directed graph representation of symbolic execution histories, the *symbolic execution graph,* which we rely upon below to combine symbolic execution with static concurrency analysis. Except for the detail of multiple paths to the same state, the reader may rely upon [HK76] for an understanding of symbolic execution graphs.

4.3 Example: Readers/writers

Symbolic execution could be used in place of static concurrency analysis to detect synchronization errors, but it would be much more expensive. On the other hand, symbolic execution is a more accurate technique. We illustrate with an example problem that static concurrency analysis alone is unable to accurately model.

The example is a mutual exclusion problem. A variable is to be shared among tasks. Any number of tasks are allowed to read the shared variable concurrently, but only if no task is currently writing to the variable. At most one task may be writing to the variable. One possible solution to this problem is to create a control task that manages access to the variable. The control task keeps track of the number of tasks currently reading the variable. Each task that accesses the shared variable is required to first obtain permission from the control task, and to afterward inform the control task that it has finished its access.

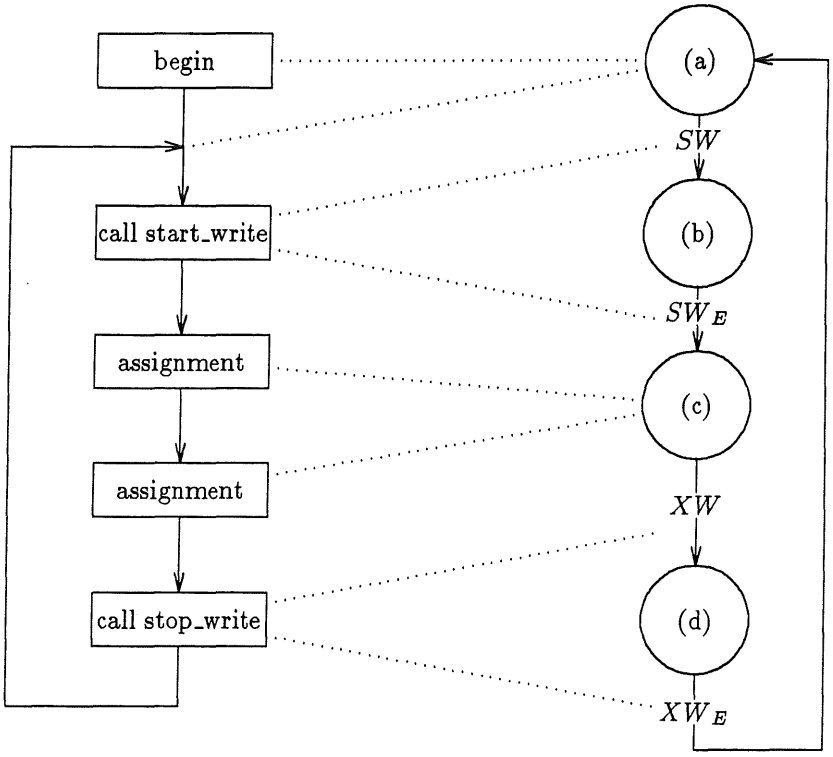
An example program illustrating this approach with one reading task and one writing task is excerpted in Figure 4.1. The control task is taken from a popular Ada textbook [Bar84, pg. 241]. Reduced representations (task interaction graphs) of the writer task and the control task are illustrated in Figures 4.2 and 4.3, respectively; the reduced representation of the reader task is identical to that of the writer.

```
task body reader is
begin
 loop
   control.start_read;
   assert(protected_variable = initial_value);
   control.stop_read;
 end loop;
end reader;

task body writer is                                          10
begin
 loop
   control.start_write;
   protected_variable := protected_variable + 1;
   --  reader must not read protected variable at this point
   protected_variable := protected_variable - 1;
   control.stop_write;
   --  initial value of protected variable is restored;
 end loop;
end writer;                                                  20

task body control is
 readers: natural := 0;
begin
 loop
   select
        accept start_write do
          while readers  >  0 loop
            accept stop_read;
            readers := readers - 1;                          30
          end loop;
        end start_write;
        accept stop_write;
   or
        accept start_read;
        readers := readers + 1;
   or
        accept stop_read;
        readers := readers - 1;
   end select;                                               40
 end loop;
end control;
```

Figure 4.1: An example program for concurrency analysis.



Abbreviations:

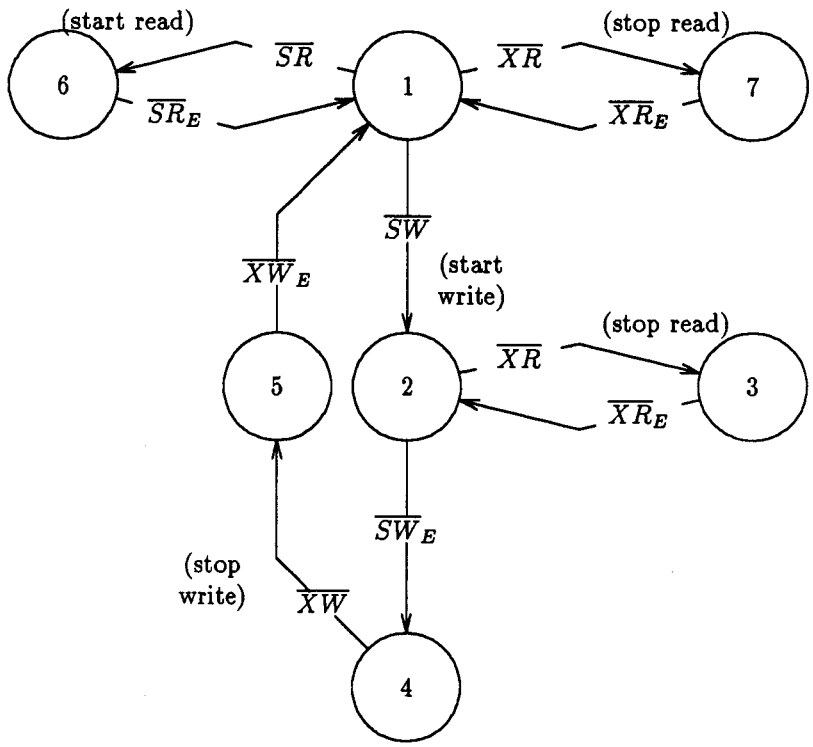
SW represents engagement of a call on entry *start_write*.

SW_E represents termination of the rendezvous at entry *start_write*.

XW represents engagement of a call on entry *stop_write*.

XW_E represents termination of the rendezvous at entry *stop_write*.

Figure 4.2: The writer task flowgraph is summarized into a state machine (task interaction graph) for analysis. Dotted lines show the relation between the two representations. Labels (a)-(d) will be used to identify nodes in subsequent diagrams and in the text. In the general case, node (a) would be split into a pair of nodes. Since there is no computation before the first iteration of of the loop, we simplify this and subsequent diagrams by letting a single node represent all the code up to the entry call on *start_write*.



Abbreviations:

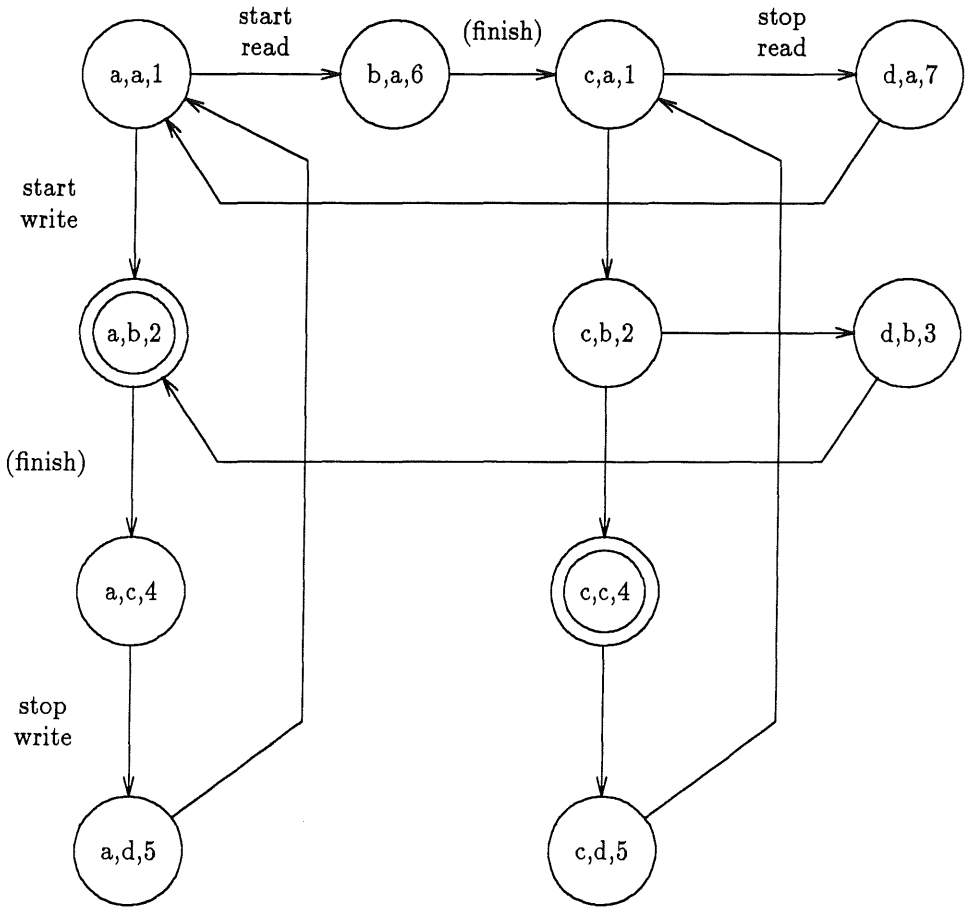
SW, XW: Entries *start_write, end_write.*

SR, XR: Entries *start_read, end_read.*

$\overline{\alpha}$: Engage an entry call (begin a rendezvous) at α, where α is one of SW, XW, SR, XR.

$\overline{\alpha}_E$: Finish a rendezvous at entry α.

Figure 4.3: State machine representation (task interaction graph) for the control task.



Progress of the reader task is shown in the horizontal direction, and progress of the writer task in the vertical. Node labels correspond to labels on the summarized flowgraph nodes in Figures 4.2 and 4.3. Error states are shown as double circles. They are:

a,b,2: Deadlock if the control task accepts *start_write* and then waits for *stop_read* when *readers*= 0.

c,c,4: Erroneous parallel access to *protected_variable* if the control tasks fails to wait for *stop_read* before terminating the *start_write* rendezvous, when *readers*> 0.

The concurrency state graph for the program in Figure 4.1, produced from the task interaction graph representations in Figures 4.2 and 4.3, is depicted in Figure 4.3. Two potential errors are represented in the graph. State (c,c,4) (that is, reader task at node c, writer task at node c, and control task at node 4) represents the possibility of illegal parallel access to the shared variable. State (a,b,2) represents a potential deadlock (even though it has an out-edge; see Section 3.3.3 in the previous chapter) because the acceptance sets of node 2 in the control task are $\{\overline{XR}\}, \{\overline{SW}\}$.

The *while* loop in the control task, with its nested rendezvous at *stop_read,* is controlled by the value of variable *readers.* Since static concurrency analysis abstracts away data values, it must account for both outcomes of the decision. This causes the two spurious errors to be included in the concurrency state graph. Deadlock results from executing the body of the loop when *readers*= 0 and the reader task cannot call *stop_read.* Simultaneous reading and writing of *protected_variable* results from failing to execute the loop body when *readers*= 1. These errors cannot actually occur, and would not be reported by symbolic execution.

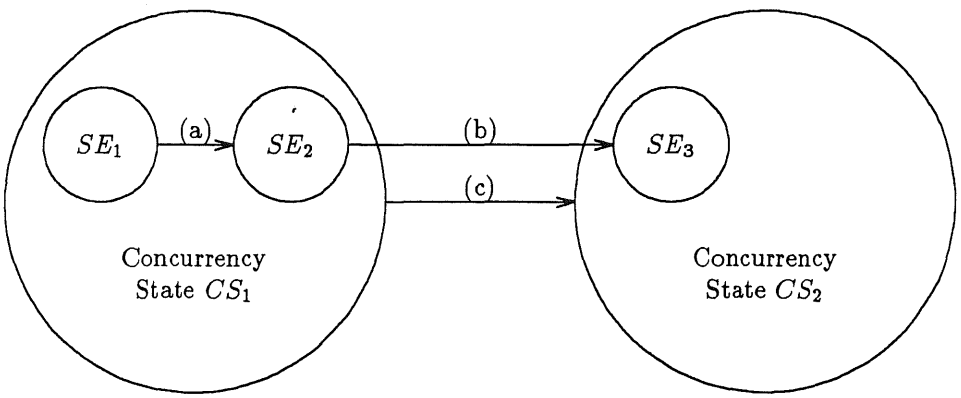
4.4 Combining the techniques

Correspondence between paths in the concurrency state graph and paths in the symbolic execution graph can be used to mitigate the problem of spurious error reports involving unexecutable program paths. In general, determining which paths are not executable is equivalent to the halting problem, but many unexecutable paths may be eliminated by detecting inconsistencies in the path condition.

4.4.1 Comparing the state spaces

Usually a directed graph of states reached during symbolic execution is not explicitly constructed, nor is the whole space explored, because in general it would be infinite. The infinite size of the state-space is a consequence of the infinite number of possible path expressions and path conditions. (Section 4.5.1 discusses loop generalization, which produces a simplified, finite representation of the state-space.) Even when no explicit representation of states and paths is maintained, the symbolic execution graph is implicit in the algorithm.

Every state in the symbolic execution graph can be associated with a corresponding node in the concurrency state graph, merely by deleting the path condition and path expression and associating nodes in the full flowgraphs with nodes in the



Edge (a) represents movement of a token through a flowgraph node that performs a non-tasking activity. The flowgraph node will be part of a region summarized into a node in the task interaction graph representation. Symbolic execution states SE_1 and SE_2 are associated with the same concurrency state CS_1.

Edge (b) represents movement of a pair of tokens through a synchronization action. Flowgraph nodes representing these actions will be represented as edges in task interaction graphs, and the joint action will be represented in the concurrency state graph. This edge in the symbolic execution graph corresponds to edge (c) from concurrency state CS_1 to CS_2.

Figure 4.4: Relation of symbolic execution states to concurrency states.

simplified flowgraphs used for concurrency analysis. The relation is many to one.[3] Edges in the symbolic execution graph can be divided into two classes (Figure 4.4): Edges representing non-tasking actions in a single task always connect two symbolic execution states that correspond to the same node in the concurrency graph. Edges representing synchronized movement of a pair of tasksalways connect symbolic execution states corresponding to two different concurrency states, and always correspond to edges in the concurrency state graph.

Thus every path in the symbolic execution graph corresponds to a path in the concurrency state graph. But the reverse is not true: some paths in the concurrency state graph do not correspond to paths in the symbolic execution graph. This is because static concurrency analysis may follow unexecutable program paths that are eliminated from the symbolic execution graph when nodes with inconsistent path conditions are discarded. In the example program (Figures 4.1–4.3), every path to the error states will induce an inconsistent predicate involving the variable *readers*. Thus the symbolic executor can determine that these error states are spurious.

[3]In general the relation can be many to many, but it is always total.

The relation between paths in symbolic execution and paths in concurrency analysis is one of *folding,* and moreover is an error-preserving abstraction with respect to a variety of synchronization errors such as illegal parallel access to variables. This justifies using symbolic execution only to refine the results of static concurrency analysis, with the assurance that additional synchronization errors will not thereby be neglected.

4.4.2 Serial application

A simple combination of concurrency analysis and symbolic execution might take the following form: First concurrency analysis is executed to completion. Nodes representing error conditions such as deadlock or possible parallel update of a shared variable are marked as "interesting." Every "interesting" node, and all ancestors of "interesting" nodes, are marked as "promising."[4] Next, symbolic execution is applied to the same program. States not corresponding to "promising" nodes in the concurrency state graph may be discarded. When symbolic execution produces a state corresponding to an "interesting" node in the concurrency state graph, that node is marked "feasible." Symbolic execution may halt when all "interesting" nodes have been marked "feasible," or when no more progress can be made down "promising" paths, or when some resource (e.g., CPU time) is exhausted.

Only nodes marked "interesting" and "feasible" need be reported to the user as probable program errors, except that if symbolic execution was cut off due to exhaustion of resources then descendants of nodes on the frontier of symbolic execution must also be reported.

When symbolic execution does reach an "interesting" concurrency state, the technique of Clarke [Cla76] can be used to provide the programmer with test data capable of exercising the erroneous path. Reproducing the error is not guaranteed by test data alone, however, due to the nondeterminism inherent in concurrent programs; a run-time scheduler that can be directed to follow a supplied concurrency history (as proposed in [Tay84b]) is also required.

[4] In a cyclic program, like the example and many process control applications, every node is likely to be an ancestor of every other, and there is no benefit in distinguishing "promising" nodes. When symbolic execution is not exhaustive, a variable estimate of "promise" (see Section 3.4) may be useful in cyclic concurrent systems. The distinction is still useful in interleaved application (discussed below).

4.4.3 Interleaved application

Often it would be more useful to receive a single error report quickly than to receive a batch of error reports after a long wait, especially if many of the reports result from a single conceptual error. In this case better performance can be realized by interleaving concurrency analysis and symbolic execution in a more tightly integrated fashion.

This method iterates phases of static concurrency analysis followed by phases of symbolic execution. A concurrency analysis phase may progress until an erroneous concurrency state is detected or until some other condition is met, e.g., until a certain number of concurrency states have been generated. Some subset of the nodes on the frontier of the search are then marked as "interesting," and their ancestors are marked as "promising." "Interesting" nodes could be chosen on the basis of a heuristic function (Section 3.4), or in the simplest case all nodes on the frontier may be deemed "interesting."

A symbolic execution phase proceeds as before, following only paths through "promising" nodes. Every concurrency state reached is marked "feasible." When symbolic execution reaches a state corresponding to a concurrency graph node with no "promising" children, execution of that path is suspended. It mustn't be abandoned completely, because it might later become "promising." Since the control point of the suspended symbolic execution can be derived from the concurrency state, it is only necessary to store the path expression and path condition to allow resumption, and to "attach" this extra information to a concurrency state. This association may be simply accomplished with a table (but note that several suspended paths could be associated with a single concurrency state). At the end of a symbolic execution phase, states on the frontier are attached to corresponding concurrency state nodes.

Subsequent phases of concurrency analysis start from those portions of the concurrency analysis frontier marked both "feasible" and "promising." Nodes that appear infeasible are not discarded, because they may be reached by symbolic execution in a later iteration, but search effort is expended only on regions of the graph most likely to correspond to executable program paths. At the end of a concurrency analysis phase, "promising" markers are recomputed and symbolic execution states attached to "promising" nodes become the new symbolic execution frontier.

A benefit of interleaved phases of symbolic execution and static concurrency analysis is that each technique may prune search by the other. From the point of view of concurrency analysis, symbolic execution is a way of pruning away infeasible execution paths. From the point of view of symbolic execution, concurrency analysis is a method of selecting paths leading to concurrency-related errors. This benefit accrues when errors are present, and analysis can be halted (and debugging begun) after

locating one or a few errors, or when the space is too large to explore exhaustively. However, exhaustive analysis of an error-free system may not benefit from pruning. At best, pruning would prevent only one state, (c,d,5), from being generated in the concurrency state graph of the example program.

4.5 Scaling up

So far, we have described symbolic execution in as straightforward a manner as possible, to simplify presentation. The previous chapter considered approaches to dealing with the combinatorial explosion encountered in static concurrency analysis. Symbolic execution introduces additional problems that must be dealt with.

4.5.1 Controlling the symbolic execution graph

Symbolic execution has combinatorial problems of its own, beyond the potential combinatorial explosion of static concurrency analysis. Since the flowgraphs used in symbolic execution are not reduced like the flowgraphs used by concurrency analysis, the algorithm may unnecessarily interleave execution of tasks. This interleaving must be avoided. A second problem is that program loops generally cause new path expressions and path conditions to be assigned at each loop iteration, making the symbolic execution graph potentially infinite. This section considers ways to avoid these two problems.

Avoiding unnecessary interleavings

The description of symbolic execution so far has assumed that each state transition moves a token through at most one node in each task flowgraph. It is better, when possible, to build up larger transition functions. The flowgraph summarizations described in Section 3.2.3 of the previous chapter, and illustrated in Figures 4.2 and 4.3, are applicable in some cases to symbolic execution as well. In particular, the number of symbolic execution states that must be enumerated can be greatly reduced if a loop in a flowgraph is logically replaced by a straight-line computation or a small set of non-looping alternatives that capture the effect of any number of loop iterations. Such a computation can be produced with the help of loop invariant conditions supplied by the programmer, or in some cases can be derived by a symbolic execution system alone.

The integration scheme described above remains valid if any part of a flowgraph is replaced by a simpler graph, provided the replaced portion does not contain any tasking-related activities such as rendezvous or task activations. In particular, loops may be generalized, and calls to non-tasking procedures can be replaced by a summary of the procedure computations. The restriction to non-tasking activities preserves the condition that every node in a flowgraph used by symbolic execution can be associated with a node in the summarized flowgraph used by static concurrency analysis. Symbolic execution states will, therefore, fall entirely within concurrency states, as before.

Access to shared variables causes no extra problems in this case. It is merely necessary to ensure that no accesses are entirely summarized away. For instance, if a shared variable were incremented and later decremented, a generalization of that sequence must still note that the variable is both read and written so that possible conflict with interleaved reads and writes by another task will be detected.

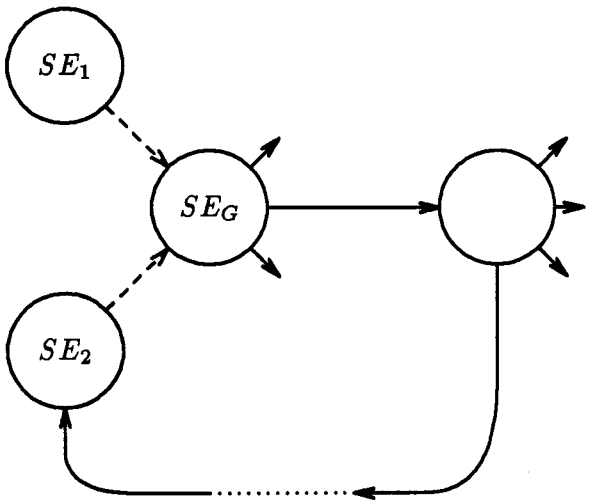
Unnecessary interleaving can be avoided even when flowgraphs are not summarized (e.g., if the system fails to determine a suitable loop invariant to cut a loop that does not contain any tasking activity). Control over interleaving equivalent to that achieved by summarizing the flowgraph can be exercised during the analysis by following execution of a single task up to the next task interaction. This approach has been taken by Knight and Grine [KG85].

Loop generalization across task interactions

Unfortunately, the restriction of generalization to non-interacting portions of task flowgraphs means that the symbolic execution graph will remain infinite in most cases, whereas generalization of all loops in sequential programs can produce finite graphs. The usual view of loop generalization is that a loop is represented by a closed-form expression or replaced by equivalent straight-line code[5] [CR84, CHT79]. That is, deriving loop expressions is a kind of *program transformation.* Because loop expressions cannot represent the effect of a loop on other tasks, nor fully capture the effect of a loop without considering the effect of other tasks on it, loop generalization across task interactions requires a slightly different view: A loop-cutting assertion is treated as a *state transformation.* The purpose of the state transformation is to map an infinite number of symbolic execution states into a finite number of representative states.

In symbolic execution of a sequential program, an inductive assertion cutting a loop maps all states associated with a particular edge in the flowgraph onto a

[5]The treatment of loop-cutting assertions in the Unisex symbolic execution system is consistent with the view advocated here



State SE_1 is replaced by the generalized state SE_G at an inductive assertion. When the loop is traversed, state SE_2 is simplified at the inductive assertion to the same state SE_G. It is then not necessary to follow this path farther.

Figure 4.5: How an inductive assertion limits search.

single representative state (see Figure 4.5). The key is that some information is discarded—some of the information in the path expression and path condition is, in effect, replaced by the assertion when it is encountered.[6] In symbolic execution of concurrent programs, each loop in each task must be cut by an inductive assertion. The set of loop-cutting assertions is a minimum requirement for keeping the symbolic execution graph finite[7]; additional assertions may be placed at other points in the task, and will be useful in reducing the number of symbolic execution states that must be explored. This *set* of local assertions (in different tasks) can be used to map an infinite set of symbolic execution states onto a finite set of representative states, even if all tasks never reach assertions simultaneously.

The symbolic execution algorithm is modified to maintain a separate path condition (PC) and path expression (PE) for each task. It is convenient to assume that conventional generalization methods have been applied to regions between task interactions, as described above, so that the flowgraphs used by symbolic execution correspond exactly to the flowgraphs used by static concurrency analysis, and each edge in the graph of symbolic execution states corresponds to an edge in the concurrency state graph. (If interleaving is controlled instead by scheduling tasks only when they interact, then the same effect may be achieved by discarding intermediate symbolic execution states between task interactions.) Assertions are associated with flowgraph edges rather than nodes. For the moment, assume that the head of each loop is cut by an inductive assertion, and that each loop contains only straight-line code. Loops may contain tasking-related actions, but may communicate values only through parameter passing during rendezvous. The restrictions on position of the inductive assertion, the restriction to straight-line code within loops, and the restriction against communication through global variables will be relaxed below.

When an assertion is encountered in a task (whether at the head of a loop or elsewhere), it is first proven from the current PC and PE of that task (else an error is reported), and then a new PC and PE are built from the assertion. Assertions immediately following inter-task communication are treated specially: the PC and PE of both tasks are temporarily conjoined for the purpose of proving the assertion. A communicated value, however, is given a new unique symbol in the PE of the receiving task, and the updated PC reflects the local assertion rather than any part of another task's PC or PE, in order to maintain independence between the PC's and

[6] An unfortunate result of this approach is that the path condition and path expression will no longer be sufficient to generate test data.

[7] When a variable value is used strictly for control, rather than computation, it may not be necessary to abstract away actual values to obtain a finite symbolic execution graph. In fact, generalizing over the possible values of such a variable may be impossible without introducing spurious error reports. This is the case with the *readers* variable in the example program. It is probably necessary to allow the programmer to "exempt" such variables from loop generalization.

PE's of different tasks. If no assertion follows a rendezvous in a given task, then any values received from another task are treated as completely unknown.

Given the restrictions above, it is clear that a path in the graph of symbolic execution states that visits a concurrency graph node twice must reach the same symbolic execution state at each visit, even if all tasks never reach an assertion at the same time. This is ensured by the independence of each task's PC and PE. The restriction to straight-line code within each loop guarantees that each task either does not move at all or else follows exactly the same subpath from the invariant assertion to its position when found in each visit to a single concurrency state. Since the PC and PE are determined completely by that sub-path (regardless of communication with other tasks), the same PC and PE must be established at each visit to a concurrency state.

Relaxing the straight-line code restriction within loops allows subsequent visits to the same concurrency state to have different associated PC's and PE's. However, there are a finite number of paths through the body of the innermost loop, so eventually a symbolic execution path involving only innermost loops must still repeat a PE and PC. This follows inductively for nested loops. Similarly, cutting the loop with an assertion somewhere other than the head may postpone but not prevent repetition of a state (and thus termination of the analysis) on every infinite path.

The number of symbolic execution states associated with a single concurrency state is at worst the product of the number of paths in each task from an assertion to a given task interaction such that each path does not pass through another assertion. This number must be finite but may be arbitrarily large, depending on the structure of a task and the placement of assertions. Fortunately, placing additional assertions in a task (for instance, directly after the end of *if* and *case* constructs) will reduce the number of symbolic execution states that must be considered. At the extreme, if an assertion is placed immediately prior to each task interaction, then a single symbolic execution state will be associated with each concurrency state.

Global variables require some special measures, since they are not described solely by the PC and PE of a single task. If the PC's and PE's of different tasks are allowed to share too much information about a global variable, an infinite symbolic execution graph can result as this information is propagated back and forth between encounters of local assertions. To avoid this, a separate PC and PE is maintained for each global variable and the variable is not mentioned in the PE of any task. When a global variable is written, its PE is a single association of that variable to the value assigned by the writing task, and its PC is copied from the PC of that task. Reading a global variable is treated just like receiving a value through a parameter during a rendezvous: The PC and PE associated with the global variable are available for proving an assertion following the assignment, but otherwise the value of the global

variable is treated as unknown. Similarly, an assertion involving a global variable is checked using the PC and PE of that variable as well as the PC and PE of the task containing the assertion. Only the PC and PE of the task are replaced by the assertion.

Practical considerations. As described, loop generalization requires placement of assertions after each rendezvous and after each read of a shared variable. Moreover, some extra control of loop generalization may be needed to prevent generalizing away important information, like the actual value of variable *readers* in the example program. As in formal verification, it may be necessary to introduce auxiliary variables in order to express suitable assertions. While these complications present no conceptual problems, they are burdensome to the programmer. A practical analysis system would relieve the programmer of some of this burden by inferring some of the trivial assertions and control information from the structure of the program. Our purpose here is limited to describing what information must be present to allow loop generalization in concurrent programs. Convenient notation and aids for providing this information are beyond the scope of this dissertation.

Partial paths

Although the description of interleaved concurrency analysis and symbolic execution considered symbolic execution of entire paths from the beginning of execution, symbolic execution of sub-paths can also be useful. Symbolic execution of a partial path begins with the path condition set to *true*, and the path expression associating unique (but unknown) values with each program variable. A path predicate describing any path that includes the partial path is built up in the usual way, and as before a path condition equal to *false* indicates an infeasible path. The implication analysis technique of Tai [TD85] is a special case in which only sub-paths of length one are considered. Symbolic execution of relatively short paths leading to an error state may be much cheaper than symbolic execution from the initial state, and may be sufficient to show the infeasibility of an error state when a sort of "handshake" between tasks is involved. It is ineffective when variable values reflect the cumulative history of task interactions. For instance, only symbolic execution rooted at the initial node of the example program can show the infeasibility of the errors in that program.

Slicing

Since the primary aim of combining symbolic execution with static concurrency analysis is to eliminate error reports resulting from infeasible program paths, it is not

necessary to develop the symbolic values of all variables. The values of interest are those that might contribute to eliminating a spurious error report, i.e., those that might be mentioned in an inconsistent path condition associated with an error state. One may reduce the size of the symbolic execution graph, as well as the complexity of each symbolic execution state, by taking a *slice* of the symbolic execution [Wei82] that develops only those values.

A slice could be specified by the programmer, but it is also possible to automatically infer a slice from a particular error state, using the method of Bergeretti and Carré [BC85]. A sliced symbolic execution would simply omit some variables from the PE, and treat those variables as completely unknown when mentioned in the PC. If a slice is inferred from an error state in the concurrency state graph, then the variables included in the PE will be a superset of those that may appear in the PC. Slicing would be especially effective when considering partial paths, since a partial path may be controlled by a "thinner" slice (i.e., the path condition of a partial path is likely to depend on fewer variables.) The result of slicing, like partial paths and generalization, is *conservative* in the sense that no feasible path will be eliminated (and hence no errors will fail to be reported), although an infeasible path may fail to be eliminated if a slice is "too thin."

4.6 Summary

Static concurrency analysis and symbolic execution complement each other in detecting anomalous synchronization patterns. Concurrency analysis reduces the number of interleavings that must be considered by the symbolic executor, while symbolic execution reduces the number of spurious error reports produced by the concurrency analysis algorithm. Opportunity for fruitful integration of these two techniques follows naturally from characterization and comparison of the state spaces they explore.

The two combinatorial problems involved in symbolic execution of concurrent programs, in addition to those shared with concurrency analysis, are unnecessary exploration of execution interleavings and the difficulty of loop generalization across task interactions. The former problem is addressed by summarizing flowgraph nodes, as in concurrency analysis, or by simulating task scheduling only at task interactions. The latter problem leads to reconsideration of what loop generalization in symbolic execution means; the state-space framework suggests a method based on transforming *states* rather than *programs.*

A similar approach to devising hybrid software analysis techniques may be useful for integration of other analysis techniques. The ultimate goal of this work is to synthesize a range of techniques for analyzing concurrent programs, exploiting the strengths of each. Simple structural analyses (and related program construction methodologies) can often prevent combinatorial explosion in static modeling, while more powerful analyses like symbolic execution and, ultimately, program proving techniques capture dynamic properties lost in the simpler techniques.

Chapter 5
Conclusions

Compromise is central to design. The ideal material for a building, or a bridge, or an aircraft fuselage, would be infinitely strong and infinitely light. No such material exists. Rather than insisting on an ideal material, engineers design the best structures they can with the materials available.

Fault detection (or prevention) for software is similar. There are no perfect techniques. Not only are there no infinitely strong and infinitely cheap techniques, there are no infinitely strong and even finitely cheap techniques. A perfect fault-detection technique is as impossible as a perfect building material.

Drawing out the analogy, we note that a bridge designer, an aircraft designer, and a building designer are likely to choose different materials. The ideal material would be the same in all three cases, but their imperfect substitutes differ in ways that make aluminum better for airplanes, concrete or steel better for buildings and bridges. Appropriate techniques for fault detection may also depend on the software artifact under construction.

Sometimes the best combination of weight and strength is not achieved by a single material. Instead, we see concrete poured into webs of steel bars to afford some combination of their advantages. Just so, an acceptable level of fault detection at an acceptable price may come not from a single technique, but from some combination of techniques with complementary characteristics. The purpose of this dissertation is to help researchers and software engineers find rational ways of devising useful combinations, particularly for detecting faults in concurrent software.

The main contributions of this dissertation are

- A classification scheme for fault detection techniques which focuses on an essential dimension of compromise,
- A precise definition of a useful relation between techniques, and a set of sufficient conditions for guaranteeing that relation, and

- A particular combination of techniques which, like steel bars in concrete, increases the strength of one without incurring the full cost of the other used in isolation.

The classification scheme presented in Chapter 1 focuses on tradeoffs between accuracy and computational effort. Accuracy must be compromised, else infinite effort is required. Exploring only a sample of the space of possible program execution states causes optimistic inaccuracy, since errors may occur in the unexplored portions of the space. Folding the state space into a smaller or more regular model of the space results generally in pessimistic inaccuracy. This perspective on the way accuracy is traded for efficiency leads to a clearer view of how techniques might be combined than the conventional dichotomy of static and dynamic analysis.

One way to combine techniques is to use a relatively cheap technique to generate candidate errors, and then to focus a more expensive and more accurate technique on those candidates, to determine which may actually occur. This approach is sound if the cheaper technique is guaranteed to lists as candidates all errors the more expensive technique would have found. Chapter 2 provides the theoretical tools to ensure this.

Chapters 3 and 4 put the theory to work. A reachability analysis technique for finding sequencing errors in concurrent programs is combined with the stronger, but much more expensive, technique of symbolic execution. Static concurrency analysis generates candidate errors, and symbolic execution filters out some errors that cannot actually occur. Because static concurrency analysis, while cheaper than symbolic execution, is still an expensive technique, the theory of error-preserving abstractions is also used to justify a parceled analysis procedure that is far cheaper at a cost of some restrictions on programs and some additional pessimistic inaccuracy.

We are still a long way from understanding software fault detection as an engineer understands building materials. Progress will depend on more theoretical studies of testing and analysis, and more empirical studies to measure the relevance of the theories. Even more, progress will depend on a perspective that treats each tool and technique not as a stand-alone solution for software quality but as a potential component of an integrated strategy for preventing and detecting faults.

Bibliography

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[ADWR86] George S. Avrunin, Laura K. Dillon, Jack C. Wileden, and William E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(2):278–292, February 1986.

[AH87] Samson Abramsky and Chris Hankin. *Abstract Interpretation of Declarative Languages.* Halstead press, New York, 1987.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachussetts, 1974.

[ALR83] American National Standards Institute. *Military Standard Ada Programming Language (ANSI/MIL-STD-1815A-1983)*, January 1983.

[Apt83] Krzysztof R. Apt. A static analysis of CSP programs. In *Proceedings of the Workshop on Program Logic*, Pittsburgh, PA, June 1983.

[Bar84] J.G.P. Barnes. *Programming in Ada.* International Computer Science Series. Addison-Wesley, Menlo Park, second edition, 1984.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[BKP84] Howard Barringer, Ruurd Kuiper, and Amir Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 51–63, 1984.

[BRV80] G. Berthelot, G. Roucairel, and R Valk. Reductions of nets and parallel programs. In W. Brauer, editor, *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 277–290. Springer-Verlag, Berlin, 1980.

[Bud81] Timothy A. Budd. Mutation analysis: Ideas, examples, problems, and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–148. North-Holland, 1981.

[CBES85] E. M. Clarke, M. C. Browne, E. A. Emerson, and A. P. Sistla. Using temporal logic for automatic verification of finite state systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 3–26. Springer-Verlag, Berlin, 1985.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CG87] E. M. Clarke and O. Grümberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 294–303, August 1987.

[CHT79] Thomas E. Cheatham, Jr., Glen H. Holloway, and Judy A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, SE–5(4):402–417, July 1979.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[CPRZ85] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London, August 1985. ACM Sigsoft.

[CR81] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation methods — implementations and applications. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 65–102. North-Holland, 1981.

[CR84] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation — an aid to testing and verification. In Hans-Ludwig Hausen, editor, *Software Validation*, pages 141–166. North-Holland, 1984.

[CRZ88] Lori A. Clarke, Debra J. Richardson, and Steven J. Zeil. Team: A support environment for testing, evaluation, and analysis. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 153–162, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

[DAW88] Laura K. Dillon, George S. Avrunin, and Jack C. Wileden. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Transactions on Programming Languages and Systems*, 10(3):374–402, July 1988.

[DGM+88] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. An extended overview of the mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 142–151, Banff, Canada, July 1988.

[Dil87a] Laura K. Dillon. Verification of Ada tasking programs using symbolic execution, Part I: Partial correctness. Technical Report TRCS 87–20, Computer Science Department, University of California, Santa Barbara, October 1987.

[Dil87b] Laura K. Dillon. Verification of Ada tasking programs using symbolic execution, Part II: General safety properties. Technical Report TRCS 87–21, Computer Science Department, University of California, Santa Barbara, October 1987.

[EFRV86] Gerald Estrin, Robert S. Fenchel, Rami R. Razouk, and Mary K. Vernon. SARA (System ARchitects Apprentice): Modeling, analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 127–140, Austin, Texas, January 1983. Association for Computing Machinery.

[EL85] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking: Branching time strikes back. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 84–96, New Orleans, January 1985. Association for Computing Machinery.

[FO76] Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8:305–330, 1976.

[FRV85] J. C. Fernandez, J. L. Richier, and J. Voiron. Verification of protocol specifications using the CESAR system. In *Proceedings of the 5th International Workshop on Protocol Specification, Testing, and Verification*, Toulouse, France, June 1985.

[FW86] Phyllis G. Frankl and Elaine J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proceedings of the Workshop on Software Testing*, pages 4–13, Banff, Canada, July 1986. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[GHL82] Steven M. German, David P. Helmbold, and David C. Luckham. Monitoring for deadlocks in Ada tasking. In *Proceedings of the AdaTEC Conference on Ada*, pages 10–25, Arlington, VA, October 1982.

[Ham87] Richard G. Hamlet. Probable correctness theory. *Information Processing Letters*, 25:17–25, April 1987.

[Hau84] Hans-Ludwig Hausen, editor. *Software Validation: Inspection, Testing, Verification, Alternatives.* North-Holland, 1984. Proceedings of the Symposium on Software Validation held in Darmstadt, FRG, September 25–30, 1983.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Founddations of Computing. The MIT Press, Cambridge, Massachusetts, 1988.

[HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol87] Gerard J. Holzmann. Automated protocol validation in *argos*: Assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, SE-13(6):683–696, June 1987.

[How77] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE–3(4):266–278, July 1977.

[How80] William E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162–169, March 1980.

[How81a] William E. Howden. A survey of dynamic analysis methods. In E. Miller and W.E. Howden, editors, *Tutorial: Software Testing & Validation Techniques*, pages 209–231. IEEE Computer Society Press, 1981. Second Edition.

[How81b] William E. Howden. A survey of static analysis methods. In E. Miller and W.E. Howden, editors, *Tutorial: Software Testing & Validation Techniques*, pages 101–115. IEEE Computer Society Press, 1981. Second Edition.

[How85] William E. Howden. The theory and practice of functional testing. *IEEE Software*, 2(5):6–17, September 1985.

[HT88] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 206–215, Banff, Canada, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[KE85] Richard A. Kemmerer and Steven T. Eckmann. UNISEX: a Unix-based symbolic executor for Pascal. *Software — Practice & Experience*, 15(5):439–458, May 1985.

[Kem82] Richard Allen Kemmerer. *Formal Verification of an Operating System Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs.* UMI Research Press, Ann Arbor, Michigan, 1982.

[KG85] John C. Knight and Virginia S. Grine. Symbolic execution of concurrent Ada programs. Technical report, Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, Virginia, 1985.

[Krö87] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1987.

[KS83] Richard B. Kieburtz and Abraham Silberschatz. Access-right expressions. *ACM Transactions on Programming Languages and Systems*, 5(1):78–96, January 1983.

[Lad79] Richard E. Ladner. The complexity of problems in systems of communicating sequential processes. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 214–223, Atlanta, Georgia, April 1979.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam88] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, April 1988.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, Pittsburgh, May 1989.

[LH83] Nancy G. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, September 1983.

[LHM+87] David Luckham, David Helmbold, S. Meldal, Doug Bryan, and M.A. Haberler. Task sequencing language for specifying distributed Ada systems — TSL-1. Technical Report CSL–TR–87–334, Computer Systems Laboratory, Stanford University, July 1987.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 96–107, New Orleans, January 1985. Association for Computing Machinery.

[LS84] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[LvH85] David C. Luckham and Friedrich W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[Mac82] John Maclean. A formal foundation for the trace method of software specification. NRL Memorandum Report 4874, Naval Research Laboratory, Washington, D.C., September 1982.

[Mar77] F. H. Martin. HAL/S - The avionics programming system for the shuttle. In *Proc. AIAA Conf. on Computers in Aerospace*, pages 308–318, Los Angeles, CA, November 1977.

[Mer74] Philip M. Merlin. *A Study of the Recoverability of Computing Systems.* PhD thesis, Department of Information and Computer Science, University of California, 1974.

[MH81] Edward Miller and William E. Howden. *Tutorial: Software Testing & Validation Techniques.* IEEE Computer Society Press, second edition, 1981.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1980.

[Mor88] Larry J. Morell. Theoretical insights into fault-based testing. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 45–62, Banff, Canada, July 1988.

[MR87] E. Timothy Morgan and Rami R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions on Software Engineering*, SE–13(10):1080–1091, October 1987.

[Myc87] Alan Mycroft. A study on abstract interpretation and 'validating microcode algebraically'. In *Abstract Interpretation of Declarative Languages*, chapter 9, pages 199–218. Halstead Press, New York, 1987.

[MZGT85] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the Ada task system by Petri nets. *Computer Languages*, 10(1):43–61, 1985.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, California, 1980.

[OF76] Leon J. Osterweil and Lloyd D. Fosdick. DAVE – a validation, error detection, and documentation system for FORTRAN programs. *Software — Practice & Experience*, 6:473–486, 1976.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[OO86] Kurt M. Olender and Leon J. Osterweil. Specification and static evaluation of sequencing constraints in software. In *Proceedings of the Workshop on Software Testing*, pages 14–22, Banff, Canada, July 1986. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[Ost84] Leon J. Osterweil. Integrating the testing, analysis, and debugging of programs. In Hans-Ludwig Hausen, editor, *Software Validation*, pages 73–102. North-Holland, 1984.

[Ost87] L. J. Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, CA, March 1987.

[Par79] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, Massachusetts, 1984.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[Pnu86] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W.-P de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, Berlin, 1986. *Lecture Notes in Computer Science* 224.

[Raz87] Rami R. Razouk. A guided tour of P-NUT. Technical Report 86–25, University of California, 1987.

[RC85] Debra J. Richardson and Lori A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, December 1985.

[RT88] D.J. Richardson and M.C. Thompson. The RELAY model of error detection and its application. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 223–230, Banff, Canada, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[RW82] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 272–278, Tokyo, Japan, September 1982.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[SC88] S. M. Shatz and W. K. Cheng. A Petri net framework for automated static analysis of Ada tasking behavior. *Journal of Systems and Software*, 1988. To appear.

[Smo84] Scott A. Smolka. *Analysis of Communicating Finite State Processes.* PhD thesis, Department of Computer Science, Brown University, 1984. Department of Computer Science Technical Report No. CS-84-05.

[Tai85] K. C. Tai. Reproducible testing of concurrent Ada programs. In *Proceedings of SoftFair II*, pages 49–56, December 1985.

[TAV86] ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee. *Proceedings of the Workshop on Software Testing*, Banff, Canada, July 1986.

[TAV88] ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee. *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, Banff, Canada, July 1988.

[Tay83a] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Tay84a] Richard N. Taylor. Analysis of concurrent software by cooperative application of static and dynamic techniques. In Hans-Ludwig Hausen, editor, *Software Validation*, pages 127–137. North-Holland, 1984.

[Tay84b] Richard N. Taylor. Debugging real-time software in a host-target environment. *Technique et Science Informatiques (Technology and Science of Informatics)*, 3(4):281–288, 1984.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

[TD85] K. C. Tai and C. Y. Din. Validation of concurrency in software specification and design. In *Proceedings of the 3rd International Workshop on Software Specification and Design*, pages 223–227, August 1985.

[TK86] Richard N. Taylor and Cheryl D. Kelly. Structural testing of concurrent programs. In *Proceedings of the Workshop on Software Testing*, pages 164–169, Banff, Canada, July 1986. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[Val88] Antti Valmari. *State Space Generation: Efficiency and Practicality.* PhD thesis, Tampere University of Technology, P.O.B. 527, SF-33101 Tampere, Finland, 1988.

[Wam85] Gordon Kent Wampler. Static concurrency analysis of Ada programs. Master's thesis, University of California, Irvine, 1985.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei88] Stewart N. Weiss. A formal framework for the study of concurrent program testing. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, pages 106–113, Banff, Canada, July 1988. ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee.

[WWFT88] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, Boston, November 1988.

[ZE88] Steven J. Zeil and Edward C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 241–248, Singapore, April 1988.

[ZS86] Pamela Zave and William Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, SE–12(2):312–325, February 1986.

Appendix A
A modular tool set to support hybrid analysis techniques in a software development environment

A.1 Introduction

This dissertation has argued that fault detection techniques for software can and should be combined. In this appendix, we consider how hybrid analysis techniques can be supported by tools in a software development environment. Tools in support of each technique must not only aid in the application of that particular technique, but also support integration by taking advantage of information obtained from, and provide further information to, other analysis techniques.

Analysis techniques and the way they may be applied in an integrated fashion are discussed in Section A.2. Requirements on a tool set to support the integrated analysis approach are also given. Section A.3 proceeds to describe CATS (Concurrency Analysis Tool Suite), which is being built to satisfy these requirements. The description of CATS proceeds from presentation of a general model for tool-supported analysis. Section A.4 details our experience with CATS through description of its application to some well-known examples. Section A.5 summarizes.

A.2 Analysis approach

Work over the past two decades has produced a number of approaches to analyzing concurrent systems, each of which has strengths but also limitations. Below, we briefly discuss some individual analysis techniques, and then propose a strategy for supporting integration of multiple techniques in a software development environment.

A.2.1 Individual analysis techniques

Formal verification. Formal verification provides a high level of assurance for a wide variety of program properties. Unfortunately, it is an expensive analysis technique even for sequential software. When concurrency is introduced, assertions involving individual processes must be augmented by auxiliary variables or control predicates involving other processes, and a cooperation or non-interference proof must be performed in addition to proofs of individual processes [OG76, Lam88].

Ideally one should obtain completely formal and machine-checked proofs of correctness for every specified property of whole systems. Unfortunately this will not be achieved in the forseeable future. It is, however, already practical to use formal verification to establish some properties of some parts of a system. Applying formal verification to a design rather than fully detailed code, or verifying critical properties or particularly critical modules (e.g., a security kernel or a safety kernel [Kem82, LH83]) are practical alternatives to attempting a complete formal verification.

One may expect that the class and scale of systems to which formal verification techniques can be applied will continue to grow, especially with tool support such as symbolic execution systems [KE85] and automated proof checkers. In the forseeable future, though, we must expect formal verification to coexist with several other analysis and testing techniques.

Testing. A basic premise of most approaches to software testing is that programs are characterized by input-output pairs, and that output is completely determined by input. In concurrent software, output is usually not completely determined by input (because of apparent nondeterminism in process schedulers) and output values may not even be the most important property of software.

Recognizing these difficulties, some researchers have formulated testing approaches suited specifically for concurrent programs. Taylor and Kelly [TK86] described a method for creating concurrency state graphs from flowgraph representations of concurrent systems. Based on these graphs, several metrics were developed to aid in structural testing of programs so represented. Tai [Tai85] described a method for reproducing sequences of synchronizations in concurrent programs to address the reproducible testing problem associated with concurrent systems. To accomplish this, he defined a sequencing control task, much like German, Helmbold, and Luckham's monitor task [GHL82], which monitors the execution sequence of tasking entry calls. Weiss [Wei88] developed a formal theory for reasoning about test coverage of concurrent programs based on representing the concurrent programs as a set of simulating sequential programs termed *serializations.*

Reachability analysis. The term "reachability analysis" is used to describe construction of a state-transition model of larger modules (or a complete system) from models of individual processes. The composite state-transition model is often called a "reachability graph." These models typically highlight synchronization structure and abstract away other details of execution. Reachability analysis has been applied to Petri nets and CSP-like and CCS-like state machine models, among others [Apt83, Tay83b, Pet81, MR87].

A primary use of reachability analysis is verification of properties of the synchronization structure of software, e.g., freedom from deadlock, freedom from starvation, and freedom from dangerous parallelism. With respect to these properties, reachability analysis provides the same level of assurance as formal verification.

Reachability graphs can also be used to support other verification and validation techniques. Each of the testing techniques mentioned above, for instance, requires an outside source of information. Kelly and Taylor's structural coverage metrics are defined in terms of a species of reachability graph. A reachability graph might also be used to generate candidate sequences for Tai's technique or serializations (which correspond to paths through a reachability graph) for Weiss's framework.

Reachability analysis suffers from two major kinds of problems. First, the details abstracted away in the simplified models may be essential to the correctness of software. Omitting these details often has the effect of producing spurious error reports. Second, the size of a global model usually grows as the product of the sizes of individual process models. Moreover, basic complexity results [Lad79, Tay83a, Smo84] imply that there is no universally applicable short-cut without further sacrificing accuracy.

A.2.2 Integrated application strategy

Given these individual detailed approaches; one can specify an integrated analysis strategy that attempts to capitalize on the strengths of the individual techniques and compensate for their weaknesses. While detailed technical descriptions of such integrated schemes have appeared in the literature [Tay84a] and in an earlier chapter of this dissertation, for our current purposes it is more helpful to first describe the gross activities and flow of information, for it is these properties that determine the necessary characteristics of a supporting tool set. Analysis at a particular point in development should include at least the following steps:

1. Examine the current structure of the system, possibly including multiple design representations, to determine which analysis techniques are currently applicable.
2. Examine a repository of asserted properties that have been established or alleged for portions of the system.

3. Produce and carry out an analysis plan based on information from the first two steps. The plan should use existing analysis results where possible, but reanalyze portions of the system if (a) A design representation for some portion of the system has been superseded by an implementation, or a more detailed design representation, necessitating a demonstration that the new representation preserves important properties of its design, (b) A portion of the system has changed, necessitating reanalysis, (c) A new requirement has been established by the user, or (d) The user has requested a greater level of assurance, e.g., a property that was previously established by simulation or testing must now be formally verified. The analysis plan should apply the cheapest techniques available to obtain the desired level of assurance. Often it will not be clear whether a particular technique is adequate, so the analysis plan will involve re-planning based on analysis results.
4. Communicate results to the user. The notion of "results" here is quite broad, and may include indications of why particular techniques could not be applied and suggestions for guiding reanalysis or restructuring the system.
5. Store results for future use. In particular, dependencies between results must be stored. For instance, if formal verification of some property of module A depends on a property of module B that has been established only by testing, then appropriate caveats must be attached to dependencies on A. Dependency management is similar to management of a network of lemmas in a formal verification system, except that we envision it encompassing properties established by a variety of means with differing degrees of confidence. Some dependencies will be obligations for yet-unwritten portions of a system.

This paradigm for analysis immediately poses several requirements on tools:

- Their domain of applicability must be explicitly stated. User guidance in applying techniques is acceptable, and in many cases will be necessary, but to the extent practical it should be possible to automatically determine applicability. For instance, an application of the Unisex symbolic execution system in verification mode depends on a program being "minimally asserted" [KE85], i.e., each loop must be cut by an invariant and each procedure must have input and output assertions. This condition would be easy to check automatically.
- If they make use of information obtained from other techniques, these dependencies must be explicitly available so that they may be accounted for in planning analysis and reanalysis.
- The results of the technique should be available in a form for storage and reuse, both within a single analysis session and for future analyses.

Tools should be modularized so that intermediate steps and results can be reused. For instance, if several tools are applied to analysis of programs in a particular language, most of them will require lexing, parsing, and static semantic analysis of source texts in that language. It is wasteful (and error-prone, if the capabilities

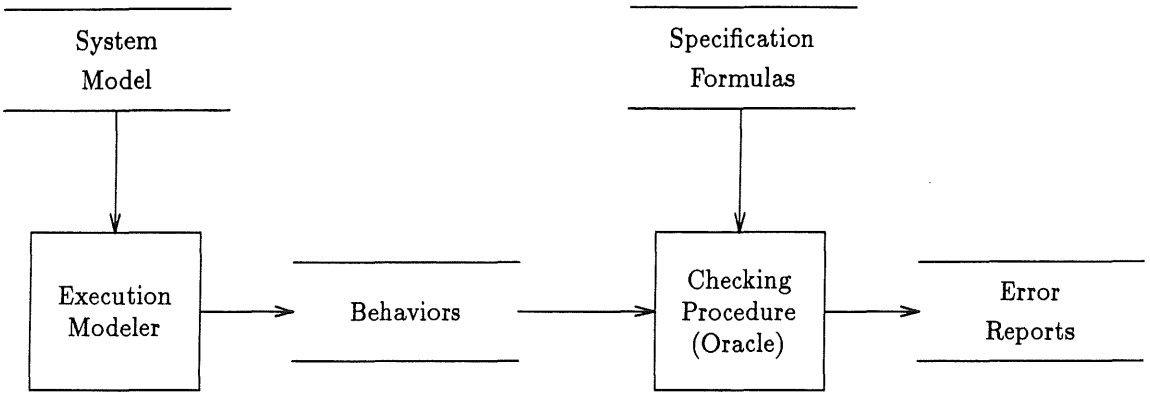


Figure A.1: Generic skeleton of a program analysis technique

are rewritten rather than reused) to perform these steps independently many times. Derivation of useful representations, whether they be parse trees or flowgraphs or reachability graphs, should be an integral part of the analysis plan.

A.2.3 General concurrency analysis model

Turning now to the more specific topic of concurrency analysis, a variety of analysis techniques can be framed in the general model that is presented in Figure A.1. An abstract representation of the system is constructed, and from this a representation of possible behaviors is constructed. This representation of possible behaviors is checked against a specification of acceptable behaviors, and violations of the specification are reported. This basic framework is general enough to describe testing (where the model is a program, the representation of behaviors is a set of actual runs, and the checking procedure is the test oracle), reachability analysis (where the representation of behaviors is a state-transition graph), and constrained expression analysis [ADWR86, DAW88] (where the representation of behaviors is a set of inequalities, and the checking procedure is a linear inequality solver), among many others.[1]

[1]With only a slight stretch of the imagination, one can even place formal verification in this framework: The model is the program, augmented by auxiliary variables or control predicates; the specification includes assertions in the program; and the representation of actual behavior consists of a set of theorems derived from the program, axiom schemata, and rules of inference. The program is accepted if each assertion in the specification also appears in the representation of actual behavior. Of course this characterization doesn't have much to do with the way people actually perform formal verification, but it may help in thinking about how formal verification can be combined with other techniques.

When reachability analysis tools are organized along these lines, they fit nicely into the integrated analysis approach described above. Analysis of the structure of a system is a prerequisite for constructing the system model, since analysis of a large system will require parceling into a set of subsystems. The specification formulas checked during analysis of one module may include not only properties explicitly asserted by the programmer, but also properties (e.g., freedom from deadlock) implicitly specified for all concurrent systems, and additional properties that are assumed in the analysis of other modules.[2] Both the representation of possible behaviors and the record of possible violations of specified behavior are information that can be made available to other analysis techniques.

The general model of Figure A.1 need not reflect the actual organization of a tool, but we will argue below that it is a good decomposition for reachability analysis tools. The following section describes a particular set of tool components that are organized along the lines of this model.

A.3 An analysis toolset for concurrent programs

The Concurrency Analysis Tool Suite (CATS) is a set of tools built according to the general concurrency analysis model just described. It is initially targeted for concurrent Ada programs. CATS was designed particularly for integration with other testing, analysis and verification tools in a software development environment. Basic design decisions for reachability analysis are discussed first, in Section A.3.1. CATS has also been influenced by its contextual software development environment, the salient features of which are noted in Section A.3.2. Section A.3.3 describes the system architecture.

A.3.1 Design considerations for reachability analysis

The design of CATS instantiates the general framework of Figure A.1 with tool components in a manner driven by lessons from previous implementations of similar systems, namely:

- Separate modeling from analysis.

[2] For instance, decomposition into "weak monitors" as described in Chapter 3 depends not only on structural analysis to ensure that dependencies among weak monitors are acyclic, but also on verifying that each weak monitor presents an interface to higher-level modules that can be modeled as simple procedure calls.

- Separate semantics of the model representation from the semantics of the actual system.

The following paragraphs discuss these rules, how they were influenced by other reachability analysis systems, and how they were applied to the design of CATS.

Separate modeling from analysis. It is tempting to tightly couple modeling (construction of a reachability graph) with analysis. For instance, one could easily combine generation of successor states with a check for deadlock, perhaps avoiding some redundant computations. An earlier prototype tool for analyzing concurrent Ada programs, constructed at UCI, did just that [Wam85]. In some other reachability analysis tools, analysis of a reachability graph is kept strictly separate from generation of the graph. In the PNut system for analysis of Petri nets [Raz87, MR87], reachability graphs are built by one program and analyzed by a completely separate program, with communication between the two only via the file system or Unix pipes. Important benefits accrue from this separation:

- The analysis component can be used with different modeling approaches. In PNut, a complete reachability graph can be generated by the Reachability Graph Builder (RGB) tool, or a single trace can be generated by the Petri net simulator. A trace is a degenerate reachability graph, so the Reachability Graph Analyzer (RGA) tool can be applied to the output of either modeling tool.
- The analysis component may be used with different models. A temporal logic model checking tool constructed at Carnegie Mellon [CBES85] has has been applied to models of sequential circuits as well as concurrent software.
- Both the modeling component and the analysis component are likely to be simpler.

This factoring of analysis from modeling need not imply a sequential two-phase operation in which all modeling precedes all analysis, although current operating system substrates make that the easiest way to compose tool components. The goal is to maintain a logical separation of modeling and analysis, and a clean interface between them, without ruling out tight integration and feedback from the analysis component to the modeling component.

Well-defined internal representations. Modeling tools for Petri nets or other simple models with simple operational semantics have obvious advantages over tools for directly modeling more complex phenomena such as concurrent software written in a language such as Ada or CSP. The usual approach to modeling software is to first build a simplified model of the complex artifact, and then to model executions

of the simplified representation. An important lesson learned from earlier prototypes noted above is that the advantages of this translation accrue only if the simplified model has an operational semantics independent of the original artifact. That is, one must be able to decompose the question

- *Does the modeling tool accurately model behaviors of the software*

into the two simpler questions

- *Is the simplified representation an accurate*[3] *representation of the original artifact,* and
- *Does the modeling tool correctly model behaviors of the simplified representation.*

If the operational semantics of the simplified representation is defined by reference to the original artifact (e.g., *"node type X represents an Ada entry call,"*) this decomposition is lost. This makes it more difficult to assure oneself that the modeling process is accurate or to diagnose the problem when it is not. Lacking an independent semantics for the simplified representation, development of the modeling tool depends on reasoning about the original artifact. When the combination of translation and modeling fails, the tool developer cannot easily localize the problem to an inadequate representation on the one hand or a failure of the modeling tool on the other.

A.3.2 Environment context

In addition to the general design considerations discussed above, the architecture of CATS is influenced by the software development environment in which it will be embedded. The *Arcadia* environment architecture and TEAM framework for analysis tools are briefly described in the following paragraphs.

Arcadia. Effective coordination of multiple analysis techniques, as described above, depends on support from a software development environment (SDE). The *Arcadia* project [TBC+88], of which CATS is part, is aimed at defining SDE architectures to support such integration with flexibility and extensibility. The *Arcadia*-1 prototype

[3] It is also necessary to say precisely what one means by "accurate." One approach is to insist that a model represent all and only the possible behaviors of a piece of software (both *accurate* and *precise,* in the terminology of [Tay83b] and [DAW88]). However, a model that meets this absolute standard cannot overcome fundamental complexity bounds (e.g., undecidability or NP-hardness), because the relation between model and original artifact amounts to a problem reduction. A reasonable approach is to insist that a model represent all possible erroneous behaviors, but allow the possibility that some impossible erroneous behaviors are also represented. It is this criterion that is defined, with sufficient conditions for ensuring it, in Chapter 2.

will demonstrate such an architecture with support for analysis, verification, and testing capabilities.

In *Arcadia*-1, software processes are themselves treated as software, conceived of as "process programs" [Ost87]. A primary constituent of the *Arcadia* environment infrastructure is a process programming language and interpreter. The operators of a process program are the tools available in the environment and the operands are the objects created by those tools and by users. The process programming framework encourages tool builders to identify small components, or tool fragments, which can be treated as operators to be composed in various ways by process programs.

In particular, the general process for concurrency analysis discussed in Section A.2.3 can be encoded and supported in the environment. Modifications to the process, based on experience, can therefore be more readily supported than otherwise, by changes to the process program and its operators. Another part of the Arcadia infrastructure, the object management system, furthers this support by providing persistence of the typed objects created, as well as the persistence of the operators and their orchestrating process program.

TEAM. Testing, Evaluation, and Analysis Medley (TEAM) is a research effort to create a framework in support of extensible integration and experimentation for automated software testing and analysis techniques [CRZ88]. The analysis capabilities of CATS are intended to fit in the TEAM framework, within the *Arcadia*-1 prototype environment.

The basic design principles leading to flexible integration of analysis support in TEAM are modularity, generic components, and language independence. These principles have had a major influence on the design of CATS. The TEAM architecture is divided into an environment support level, language processing support, basic analysis components and advanced testing capabilities. The CATS system as a whole is an instance of an advanced testing capability. It utilizes the environment support level (e.g., for persistence of intermediate results), the language processing tools (for producing simplified models of software to be analyzed), and basic analysis components (primarily a symbolic interpretation system).

A.3.3 CATS architecture

The organization of the CATS system, and the tool components that comprise the system, fits the general model of Figure A.1. A more detailed diagram of information flow in the CATS system is presented in Figure A.2. The system model is a set

of task interaction graphs (TIG's) [LC89] derived from the source code of a concurrent Ada program or other design notation for rendezvous-style concurrent systems. A compiler front end produces a semantically-analyzed graph representation of the program, and this representation is then translated into TIGs. An exhaustive enumeration of all possible sequences of interactions among tasks is represented by a global state-transition graph, which we call a task interaction concurrency graph (TICG). Explicit specifications in a branching-time propositional temporal logic are checked using the decision procedure described by Clarke, Emerson, and Sistla [CES86], and a separate procedure verifies freedom from deadlock. Behaviors that violate the specification formulas may be reported directly to the user, or may first be refined by attempting symbolic execution of a corresponding path through the program as described in Chapter 4.

Task Interaction Concurrency Graphs

Possible behaviors of a set of task interaction graphs are represented by a state-transition graph, called a task interaction concurrency graph (TICG). The nodes of the TICG are tuples of task interaction graph nodes, representing reachable states of the system (or subsystem) under analysis. Edges in the TICG represent interactions between individual processes (tasks).

In general, the size of a task interaction concurrency graph may be the product of the sizes of the individual task interaction graphs from which it is constructed. Exhaustive global analysis of large systems can never be practical, so parceling of analysis is an absolute requirement. The granularity to which systems must be divided for analysis depends on space efficiency, fast traversal, and locality of reference in the representation of TICG's. A reasonable goal is that the maximum practical granule size should be large enough that, when a system is divided into modules in design, each natural module can be analyzed in whole. This requires graphs representing thousands of states and events to be constructed and analyzed in a few minutes.

In addition, the TEAM framework suggests an organization that clearly separates the generic aspects of reachability graphs from those aspects of TICG's dependent on the language of the software under analysis (e.g., Ada). As it happens, a design that isolates language-dependent features also serves the efficiency needs of reachability analysis. For these reasons, task interaction concurrency graphs in CATS are divided into two parts: A language-independent attributable graph structure, and a set of attributes.

The graph structure underlying a TICG contains no attributes except for the connectivity of the reachability graph. The attributable graph structure is completely independent of whether the reachability graph is built from task interaction graphs,

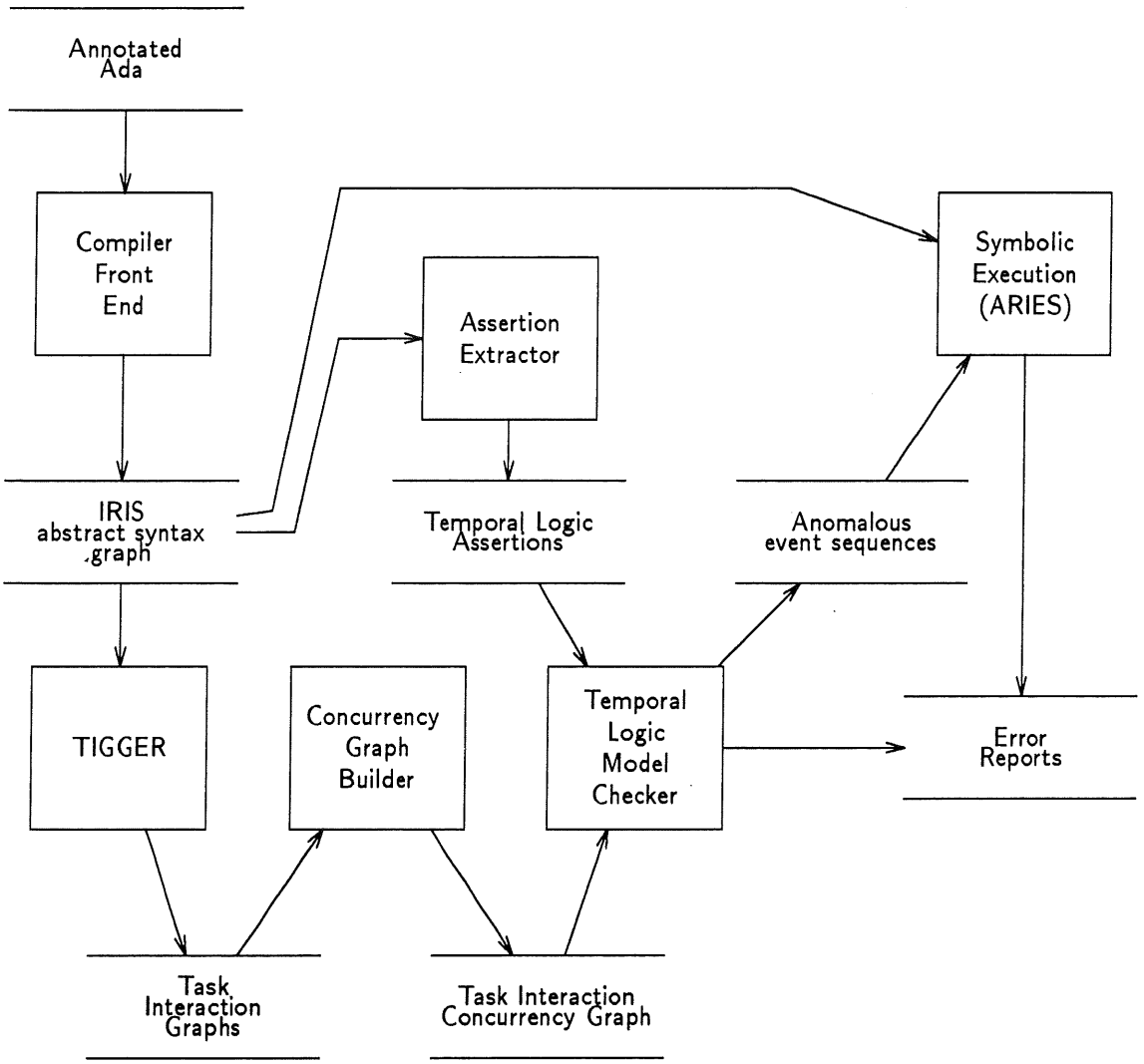


Figure A.2: Information flow among tool components in CATS.

a Petri net, or some other model. States and events are represented as attributes of nodes and edges, respectively, of the basic attributable graph structure. Attributes are encapsulated separately from the graph structure.

For space and time efficiency, a parallel array structure is used for representing the graph structure and its attributes (see Figure A.3). This results in good locality of reference, since a typical reference pattern is to traverse the whole graph while accessing only one or two attributes. In particular, checking temporal logic specification formulas involves a traversal for each subformula, and in each traversal one or two boolean attributes is accessed and another is produced. The efficiency and convenience of the parallel array representation comes at some cost in abstractness: The representation of state and edge identifiers as a contiguous set of small integers is not fully hidden.

An interesting example of reuse of the generic attributable graph structure underlying TICG's is a special-purpose, internal representation of task interaction graphs. Just prior to constructing a TICG, a very optimized internal representation of task interaction graphs is built (with, e.g., all identifiers of tasks and entries replaced by small integers), and references to the original structure are treated as attributes of the graph. The original structure of TIG's reflects a different set of design tradeoffs than reachability graphs: Task interaction graphs of individual tasks are much smaller than TICG's, so compromising abstraction for performance would be inappropriate. They are instead represented by an attributed graph structure automatically generated by *P-Graphite* [WWFT88]. *P-Graphite* manages persistence of TIG's, allowing them to exist beyond the lifetime of a single program without explicit input/output. The last-second translation of this representation into optimized form allows CATS to increase performance without losing all the benefits of the *P-Graphite*-based representation.

Checking sequencing constraints

CATS checks two kinds of sequencing conditions on the behavior of concurrent software as modeled by a task interaction concurrency graph. Freedom from deadlock is considered an implicit specification, and is checked using a special-purpose procedure. Additional constraints can be explicitly specified by the user, by embedding temporal logic assertions in the software. Temporal logic assertions are checked using an adaptation of the model checking algorithm of [CES86]. When a sequencing error is detected, example violating behaviors (sequences of events) are produced. These violating behaviors are currently used only for reporting errors to the user; in the future they will first be checked for feasibility using symbolic execution.

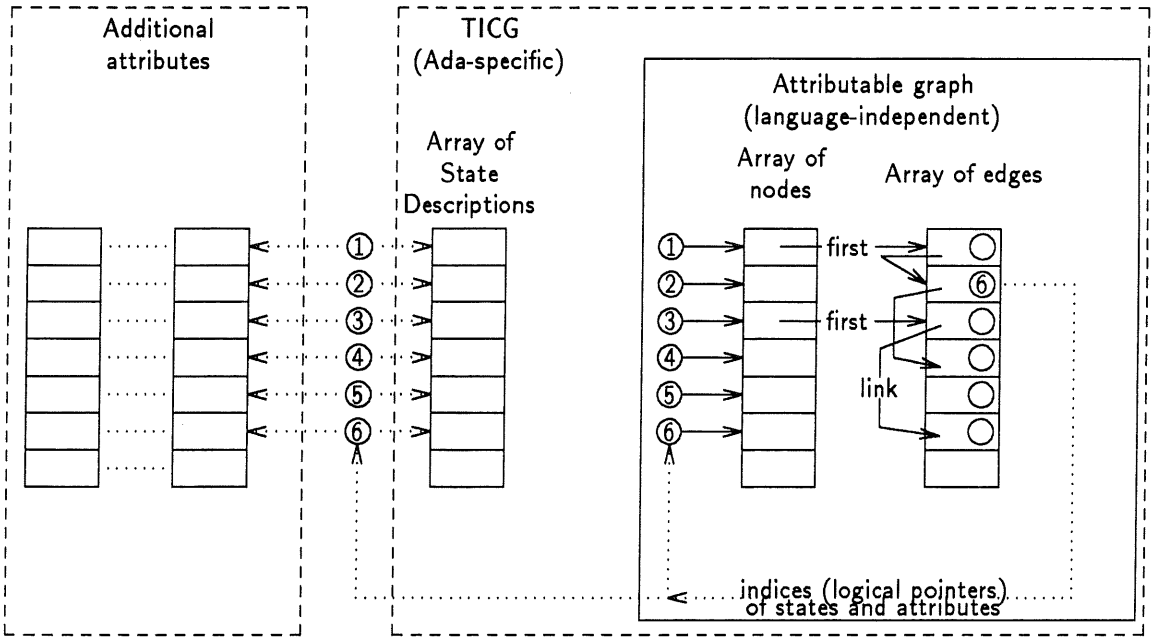


Figure A.3: Division of task interaction concurrency graphs (TICG's) into a language-independent attributable graph structure and a set of attributes, using a parallel array structure for efficiency and to allow easy creation of new attributes. For simplicity, only attributes of states (nodes) are shown here; edges (events) are attributed in a similar manner.

Temporal logic assertions. The temporal logic supported by CATS is a propositional, branching time logic based on the computation tree logic (CTL) of [CES86]. But whereas the propositions in a CTL formula refer to states (nodes of a reachability graph), it is often more convenient to describe sequences of events. Temporal logic assertions embedded in Ada programs must describe tasking events, which are represented by edges in a TICG. The CTL logic has therefore been extended to have the following logical structure:

⟨*assertion*⟩ ::= ⟨*temporal-formula*⟩

⟨*temporal-formula*⟩ ::= temporal operator {⟨*temporal-formula*⟩}

⟨*temporal-formula*⟩ ::= boolean operator {⟨*temporal-formula*⟩}

⟨*temporal-formula*⟩ ::= temporal operator {⟨*event-formula*⟩}

⟨*event-formula*⟩ ::= ⟨*event-description*⟩

⟨*event-formula*⟩ ::= boolean operator {⟨*event-formula*⟩}

A temporal logic formula describes a set of states in a reachability graph in which some property holds true. Formulas of temporal logic may alternatively be referred to as state formulas. An event formula describes a set of edges in a reachability graph on which some event holds true. Event formulas may be operands to temporal formulas, but the result of the operation is always a state formula. Boolean operators may be applied either temporal or event formulas, but cannot mix the two, i.e., it is meaningless to combine event and state descriptions with a boolean operator. The result of a boolean operator is always the same type as its operand(s).

Model checking. As in the design of the reachability graph representation, a main objective in designing the temporal logic checking component of the system was to keep all language specific aspects isolated from the rest of the model checking system. This is accomplished by separating the processing of atomic propositions, which describe individual events (edges) and states (nodes), from the logical connectives that are independent of the attributes associated with nodes and edges (see Figure A.4).

The atomic propositions of each subexpression in a temporal logic assertion are patterns that match attributes of individual states or edges in a task interaction concurrency graph. These descriptions are the only part of the temporal logic that depend on the language-dependent aspects of task interaction concurrency graphs. A single module, isolated from the rest of the model checker, is responsible for determining which nodes and edges match a description. The current version matches descriptions of Ada task interactions. If the model checking component of CATS were

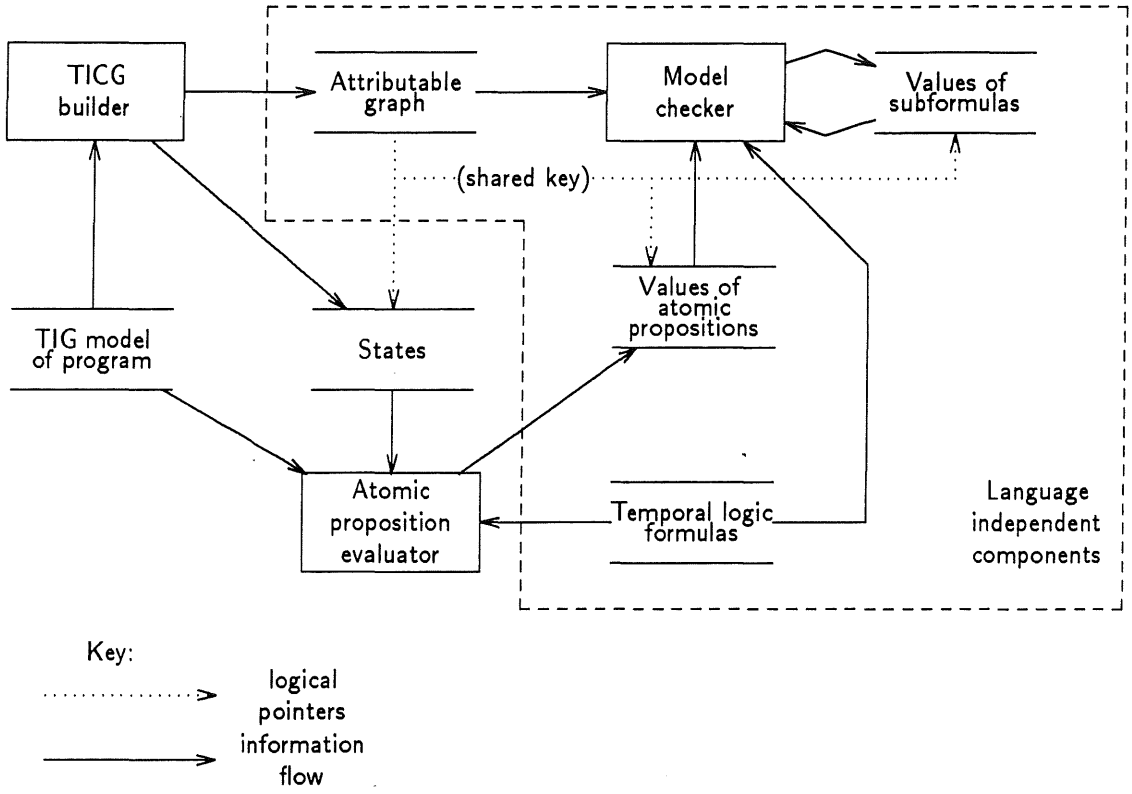


Figure A.4: Many of the components of CATS are independent of the particular system model. In the initial version, the language-dependent parts (outside the dashed line) model concurrent Ada programs. The model-independent portions can be reused for other reachability analysis capabilities, e.g., of executable design notations. The dotted lines represent the logic pointer relation illustrated in Figure A.3. Values of atomic propositions and assertion subformulas are boolean attributes of the graph structure.

to be used for a different language or model (say, Petri nets), then a suitable procedure for checking atomic propositions (e.g., markings of the net) would have to be written.

Each temporal logic assertion is parsed into an intermediate tree structure, where the internal nodes correspond to logical connectives (including the temporal operators) and external nodes are atomic propositions. An assertion is verified by interpreting the tree bottom up, associating with each node a boolean attribute to indicate the value of that subexpression in each node (or edge) of the reachability graph. The overall procedure dispatches to the language-dependent portion of the model checker to evaluate atomic propositions. Logical and temporal connectives are interpreted by other modules, which depend only on the structure of the reachability graph (the language-independent attributable graph structure).

Temporal logic assertions are embedded in the users' source code. Each such location in a program may be represented by many states in the TICG. The set of TICG states corresponding to the location of a single assertion is termed the "context" of the assertion. A relation is maintained between an assertion and its context, and a sequencing violation is reported only if a temporal assertion evaluates to *FALSE* in the context of the assertion. Context checking, along with checking atomic propositions, makes up the language-dependent portion of the model checker.

A.4 Example applications

Both CATS and the environment into which CATS will eventually be placed are currently under construction. Many of the major components of CATS are already operational, and we have applied these to some well-known problems.

What works, what doesn't. The first stage of CATS is a translation from source code to TIG models. This translation utilizes a compiler front-end that is shared with a variety of analysis capabilities in the TEAM framework. Currently, a parser for Ada is operational, but static semantic analysis is still in development, as is the translation from the semantically-analyzed intermediate form to TIG structures. In lieu of automatically generated TIG structures, the currently operational reachability graph (TICG) generator accepts hand-coded descriptions of TIG's. A temporal logic model checking procedure is also operational, as are facilities for extracting temporal logic assertions from the internal representation of a program. A special-purpose procedure for detecting deadlock is also operational. The relation between assertions, source code, and TIG's, which will eventually be maintained automatically, is currently simulated by a correspondence between identifiers in TIG's and an intermediate

to be used for a different language or model (say, Petri nets), then a suitable procedure for checking atomic propositions (e.g., markings of the net) would have to be written.

Each temporal logic assertion is parsed into an intermediate tree structure, where the internal nodes correspond to logical connectives (including the temporal operators) and external nodes are atomic propositions. An assertion is verified by interpreting the tree bottom up, associating with each node a boolean attribute to indicate the value of that subexpression in each node (or edge) of the reachability graph. The overall procedure dispatches to the language-dependent portion of the model checker to evaluate atomic propositions. Logical and temporal connectives are interpreted by other modules, which depend only on the structure of the reachability graph (the language-independent attributable graph structure).

Temporal logic assertions are embedded in the users' source code. Each such location in a program may be represented by many states in the TICG. The set of TICG states corresponding to the location of a single assertion is termed the "context" of the assertion. A relation is maintained between an assertion and its context, and a sequencing violation is reported only if a temporal assertion evaluates to *FALSE* in the context of the assertion. Context checking, along with checking atomic propositions, makes up the language-dependent portion of the model checker.

A.4 Example applications

Both CATS and the environment into which CATS will eventually be placed are currently under construction. Many of the major components of CATS are already operational, and we have applied these to some well-known problems.

What works, what doesn't. The first stage of CATS is a translation from source code to TIG models. This translation utilizes a compiler front-end that is shared with a variety of analysis capabilities in the TEAM framework. Currently, a parser for Ada is operational, but static semantic analysis is still in development, as is the translation from the semantically-analyzed intermediate form to TIG structures. In lieu of automatically generated TIG structures, the currently operational reachability graph (TICG) generator accepts hand-coded descriptions of TIG's. A temporal logic model checking procedure is also operational, as are facilities for extracting temporal logic assertions from the internal representation of a program. A special-purpose procedure for detecting deadlock is also operational. The relation between assertions, source code, and TIG's, which will eventually be maintained automatically, is currently simulated by a correspondence between identifiers in TIG's and an intermediate

(postfix) form of assertions. And while experiments with a symbolic interpretation system have begun, integration of symbolic execution into the operational system is still in the design stages; one of the experiments described below simulates the contribution of symbolic evaluation to concurrency analysis.

The following experiments were performed with the currently operational portions of CATS. In other words, we produced a TIG representation of a concurrent program by hand, along with temporal logic assertions. The remainder of the processing — building the TICG, checking it for deadlock and for conformance with temporal logic formulas, and generation of example behaviors — was automatic.

Dining philosophers

The dining philosophers problem is a well-known example of exposure to deadlock. It is attractive for experimentation primarily because so many other developers of tools and techniques for analyzing concurrent software have used it, facilitating comparisons. Moreover, it is a simplified version of a significant class of problems for concurrent systems. We assume the reader has some familiarity with the problem.

Experiment 1: Capacity and performance. The first version of the dining philosophers considered is the classic (non-)solution, in which all n philosophers remain seated at the table and each picks up the left fork before the right. In this version deadlock occurs when each philosopher holds one fork. In this case analysis is trivial, once the TICG is built. The interesting questions concern capacity and performance of tool components, which will be important in determining the granularity of modules that must be achieved for practical analysis of large systems.

The dining philosophers system consists of $2n$ processes (Ada tasks), one for each philosopher and fork. Using an unoptimized representation of the tasks, in which each communication (fork up, or fork down) is represented by two distinct interactions (beginning and end of Ada rendezvous), the performance of the TICG builder, compiled under Verdix Ada 5.41 and measured on a Sun 3/260, are as follows:[4]

[4]These are approximate elapsed times, obtained from the Unix `time` command, and were not obtained under controlled conditions. Variations ranging up to a factor of 2 were observed. Times in tables are always for building the TICG, and do not include checking for deadlock or violations of temporal logic formulas. In every case, the analyses were much faster than building the TICG.

Philosophers	Tasks	States	Edges	Minutes (approx)
2	4	40	56	< 1
3	6	268	576	< 1
4	8	1792	5168	1
5	10	11,744	42,440	7

This experiment suggests that a reasonable granularity is in the neighborhood of 8 processes, for processes with simple synchronization structure. While global analysis of large systems will certainly not be feasible, they can be analyzed efficiently if broken into modules of no more than 6 or 8 processes.

The times given above include only building the TICG. For the 4-philosopher problem, deadlock checking took half a minute and produced a report of a sequence of task interactions leading to the single deadlocked state:

```
Deadlock Violation Detected:
    Engage phil1, fork1.up
    Finish phil1, fork1.up
    Engage phil2, fork2.up
    Finish phil2, fork2.up
    Engage phil3, fork3.up
    Finish phil3, fork3.up
    Engage phil4, fork4.up
    Finish phil4, fork4.up
phil1  attempting to engage fork4.up
phil2  attempting to engage fork1.up
phil3  attempting to engage fork2.up
phil4  attempting to engage fork3.up
fork1  attempting to accept fork1.down
fork2  attempting to accept fork2.down
fork3  attempting to accept fork3.down
fork4  attempting to accept fork4.down
```

Experiment 2: Resource ordering. The dining philosophers experiment was repeated, with the modification that an ordering is imposed on forks and each philosopher picks up its lower-numbered fork first. This is a well-known approach to deadlock avoidance, and had the expected effect: The deadlocked state that is detected in the first version of the dining philosophers is absent in the version with resource ordering.

The size of the TICG, time to generate it, and time to check each state for deadlock are approximately the same as for the first version of the dining philosophers problem.

Temporal logic checking. After freedom from deadlock was verified in the four-philosopher system with resource ordering, the following temporal logic assertion was checked:

```
(always (eventually accept fork1.up) and
        (eventually accept fork2.up) and
        (eventually accept fork3.up) and
        (eventually accept fork4.up))
```

Together with the FIFO acceptance of queued entry calls guaranteed by Ada, this property should guarantee that philosophers never starve. If the task scheduler is unfair, though, the property may not hold. A sequence of events violating the constraint is reported to the user:

```
    Engage phil1, fork1.up
    Finish phil1, fork1.up
    Engage phil1, fork4.up
<<LOOP>>
        Finish phil1, fork4.up
        Engage phil1, fork1.down
        Finish phil1, fork1.down
        Engage phil1, fork4.down
        Finish phil1, fork4.down
        Engage phil1, fork1.up
        Finish phil1, fork1.up
        Engage phil1, fork4.up
```

A version of the model checker that will allow the user to assume fair scheduling (using the extended CTL algorithms of [CES86]) is under development.

Experiment 3: Butler. Another well-known solution to the dining philosophers problem is the addition of a "butler" process, which ensures that the number of philosophers at the table is one fewer than the number of forks. The butler task is as follows:

```
task body butler is
  room_occupants: natural := 0;
  room_limit: constant natural := problem_size - 1;
 begin
  loop
       select
         when room_occupants < room_limit =>
         accept enter;
         room_occupants := room_occupants + 1;
       or                                                    10
         accept leave;
         room_occupants := room_occupants - 1;
       end select;
  end loop;
 end butler;
```

Each philosopher is modified to enter the room before picking up either fork, and to leave the room after eating. When the number of philosophers in the room reaches the limit (one less than the number of forks), the butler refuses to let another philosopher enter until some philosopher leaves, thus avoiding the deadlock situation.

Addition of the butler task, combined with additional complexity in each philosopher, adds considerably to the size of the generated TICG. With four philosophers (9 tasks), a table overflows and causes the analysis procedure to terminate after 35 minutes of processing.

Philosophers	Tasks	States	Edges	Minutes (approx)
2	5	139	250	< 1
3	7	1681	4188	< 1
4	9	*	*	> 35

The dining philosophers problem with a butler illustrates one of the problems of concurrency analysis, namely, that it abstracts away variable values even when those values are critical to the synchronization structure of a system. In the present case, abstracting away the variable `room_occupants` causes an inaccurate representation of the behavior of the butler — it fails to prevent all the philosophers from simultaneously entering the room. When the TIG model of dining philosophers with a butler was analyzed, it reported a deadlock that cannot actually occur.

Experiment 3a: Unrolling the butler. A partial solution to the problem of spurious error reports is to combine static concurrency analysis with symbolic execution. Elsewhere in this dissertation, we have described an approach in which candidate errors exposed by static concurrency analysis are used to guide a symbolic executor.

CATS is designed to support this approach, as indicated in Figure A.2. However, neither the symbolic execution capabilities of the ARIES generic interpretation system [ZE88] nor the remainder of CATS is yet mature enough to test this combination.

An alternative way to combine symbolic execution with static concurrency analysis is to "unroll" the butler. Every programmer knows that complexity of control flow can often be traded for flags and counters. Symbolic execution can be used to automate a translation from flags or counters to control flow, similar to loop unrolling in an optimizing compiler. (This is actually a slight generalization of unrolling definite loops, which was proposed by Taylor in [Tay83b]). For a fixed value of `room_limit`, the butler task can be unrolled into:

```
loop
    select -- room_occupants = 0;
      accept enter;
      select -- room_occupants = 1;
        accept enter;
        select -- room_occupants = 2;
            accept enter;
            -- Etc. until room_occupants = room_limit
        or
            accept leave;                                    10
        end select;
      or
        accept leave;
      end select;
    or
      accept leave;
      -- ERROR: room_ocupants = -1
    end select;
end loop;
                                                             20
```

The count of philosophers in the room is replaced by nesting copies of the loop body. At each step in the unrolling, the guard predicate `room_occupants < room_limit` evaluates either to *TRUE* or *FALSE,* and can be discarded. When it evaluates to *FALSE,* the unrolling process terminates (the innermost copy of the `select` clause contains only the `accept leave` alternative). The eventual termination of the unrolling process guarantees that the value of `room_occupants` will never exceed a fixed maximum value, but information in the butler task alone is not sufficient to verify that it cannot be decremented after reaching zero. For this reason, the outermost `accept leave` alternative is associated with an error state in the TIG representation; absence of this state in modeled executions can be checked automatically.

A TIG representation of the unrolled version of the butler was constructed, and analysis was repeated with this version. Note that, even though the unrolled version of butler is larger than the rolled version, it does not cause the TICG to become larger. This will always be the case when an eliminated variable was used to keep track of the states of other processes, since the value of the variable is purely a function of the states of other tasks.

Philosophers	Tasks	States	Edges	Minutes (approx)
2	5	56	59	< 1
3	7	932	1856	< 1
4	9	12,694	35,615	5

Given a fair task scheduler, each philosopher will have an infinite number of opportunities to eat provided the butler infinitely often lets a philosopher into the room. The assertion `(always (eventually accept butler.enter))` was verified by the temporal logic model checker. For the four philosopher problem, building the TICG and checking the assertion were completed in under 6 minutes of elapsed time.

A.5 Summary

Effective support for analysis, verification, and testing requires coordinated application of a variety of techniques. As the difficulties of analyzing concurrent software are especially severe, so it is especially important that multiple techniques be brought to bear on the problem. The approach advocated here allows each technique to make use of information gathered by other techniques.

CATS is designed to be integrated with other tools and techniques in a software development environment. It requires structural information to allow decomposition of large systems for independent analysis, and it produces information useful in other techniques, particularly symbolic execution and testing. In keeping with the TEAM framework [CRZ88] and the *Arcadia* environment architecture [TBC+88], CATS is composed of small tool fragments to maximize flexibility and reuse. Language-specific and language-independent components are clearly separated, and major parts of each are reusable.

Both CATS and the *Arcadia*-1 environment into which CATS will be placed are still in development. Experience with the operational portions of CATS make us optimistic about the role static concurrency analysis can play in combination with other techniques for analyzing concurrent software.